Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Memo 385 December 1976 Logo Memo 32

Parsing Protocols Using Problem Solving Grammars

Mark L. Miller and Ira P. Goldstein

A theory of the planning and debugging of programs is formalized as a context free grammar. The grammar is used to reveal the constituent structure of problem solving episodes, by parsing protocols in which programs are written, tested and debugged. This is illustrated by the detailed analysis of an actual session with a beginning student. The virtues and limitations of the context free formalism are considered.

This report is a revised version of the second half of AI Working Paper 122 (Logo Working Paper 50). It describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This research was supported in part by the National Science Foundation under grant C40708X, and in part by the Division for Study and Research in Education, Massachusetts Institute of Technology.

Table of Contents

Acknowledgements

Thanks are due to H. Abelson and H. Peelle for carefully criticizing an earlier version of this paper. The authors would also like to thank Carol Roberts for assistance with the illustrations.

1. A Planning Taxonomy

> This study ... forms part of an attempt to construct a formalized general theory ... and to explore the foundations of such a theory. The search for rigorous formulation ... has a much more serious motivation than mere concern for logical niceties or the desire to purify well-established methods of ... analysis. Precisely constructed models ... can play an important role, both negative and positive, in the process of discovery itself. By pushing a precise but inadequate formulation to an unacceptable conclusion, we can often expose the exact source of this inadequacy and, consequently, gain a deeper understanding of the ... data.
>
> [Chomsky 1957]

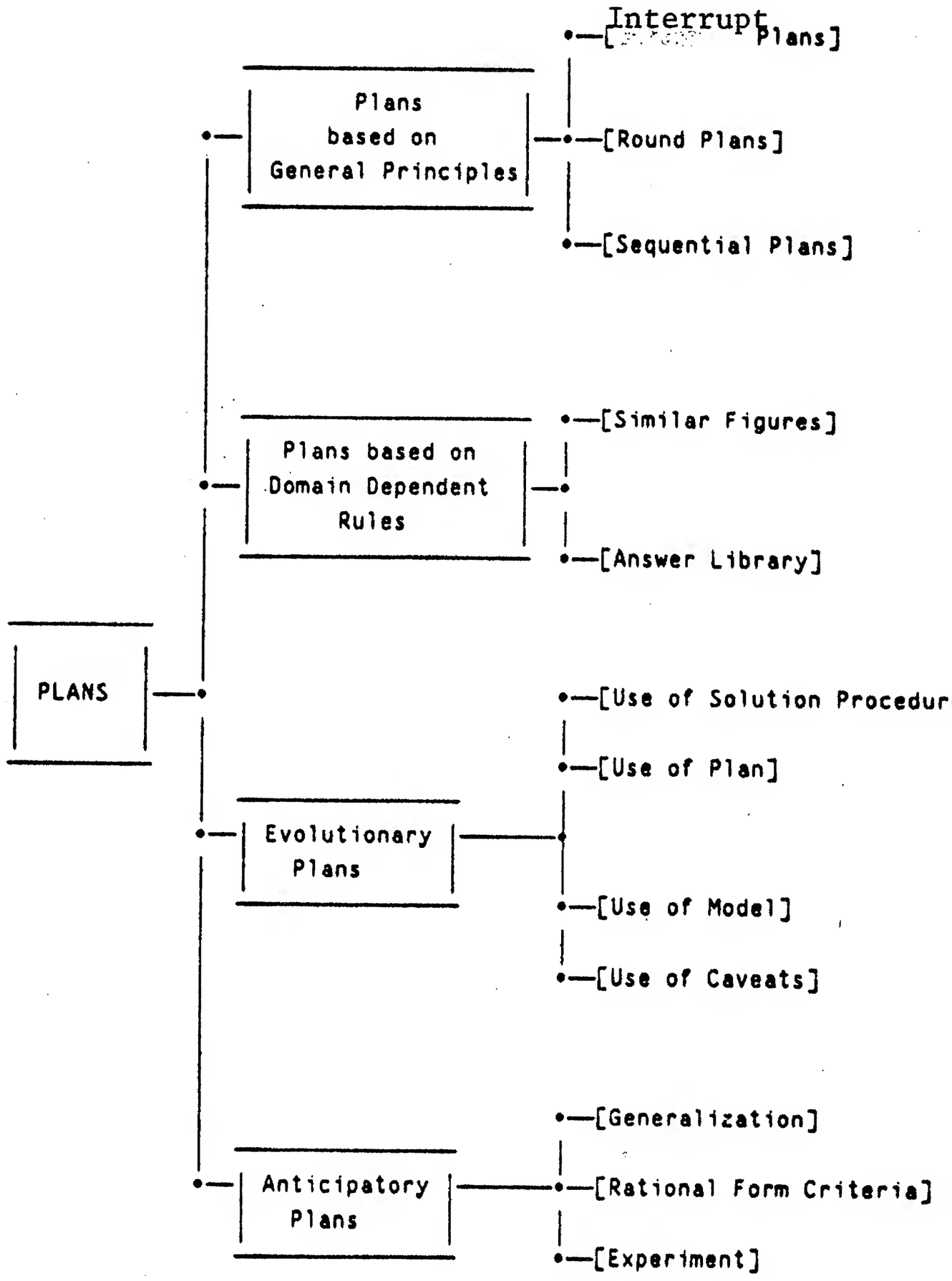
In this paper, we explore the use of a grammar to parse problem solving protocols for the purpose of revealing more clearly their constituent structure. We view problem solving as an alternating sequence of planning and debugging episodes. In the next section, we develop a taxonomy of plans, i.e. problem decomposition techniques. Section 3 formalizes the decisions involved in choosing among alternative methods in this taxonomy by means of a grammar. Section 4 develops a complementary grammar for debugging. Both these grammars are then applied in sections 5 and 6 to parse a protocol of a student engaged in an elementary graphics programming project.

Our planning theory derives from the common observation that the fundamental technique for solving a seemingly intractable problem is to divide it into more manageable subproblems. A critical question for the aspiring problem solver then becomes, "*on what basis* shall I decompose this problem into subproblems?". *Plans* are strategies for performing this decomposition.[1]

{Figure 1} illustrates part of our planning taxonomy. The top level classification is into four mutually exclusive categories:

(1) General Principle Plans
(2) Evolutionary Plans
(3) Anticipatory Plans

A Taxonomy of Planning



{Figure 1}

(4) Domain-dependent Plans.[2]

These four categories reflect the intuition that guidance for a problem solver can come from only four directions: (s)he can look upwards to general principles, downwards to the specifics of the domain, backwards to past solutions, or forwards to anticipated future difficulties. There are interactions: A past solution can be useful simply as a concrete embodiment of a general principle; domain dependent plans can range in specificity from the very general to the very specific. Nevertheless, division into mutually exclusive categories serves as a useful first approximation. The remainder of this section defines each category in greater detail.

The first category consists of Plans based on *General Principles* having wide applicability to many domains. Perhaps the most important member of this class is the *Sequential Plan* which attempts to structure the problem into independent pieces. Other important General Principle Plans which we shall not define here but whose names are indicative of their nature are: Recursive Plans, Interrupt Plans, Search Plans and Timesharing Plans. The strength of General Principle Plans is their wide applicability: their weakness is the abstractness and hence potential vagueness of their advice.

Evolutionary Plans supplement and constrain these General Principle Plans by suggesting that the critical problem decomposition be made on the basis of an analogy to a previously solved problem. Evolutionary Plans are particularly useful when they are applied to *projects*, i.e. sequences of problems, one building upon the next towards some ultimate goal.[3]

Anticipatory Plans constrain the form of the problem decomposition in order to ensure that the solution will be extensible and modifiable. They anticipate potential goals by applying aesthetic criteria to the structure of the problem solving process. The use of variables for generalization and subprocedures for modularity are two examples of Anticipatory Planning applicable to the programming domain. Anticipatory Plans involving preliminary experimentation interact closely with Evolutionary Plans when the experimentation involves first solving an auxiliary problem in order to try out possible consequences, followed by evolutionary use of the knowledge acquired in a subsequent attack upon the original problem.

Domain Dependent Plans form a fourth class. These Plans structure the solution procedure in a manner which depends on properties specific to the particular domain. Polya's *Pattern of Similar Figures* (a plan for solving geometry construction problems) is one such example [Polya 1962]. The use of a domain dependent *Answer Library* is another. Descriptions of the

applicability conditions for domain dependent methods provide the basis for matching problems to solutions. This plan type is the framework for *pattern directed invocation*, an organizational scheme for problem solving central to systems such as Planner [Hewitt 1972] and Hacker [Sussman 1973].

Complete descriptions of planning behavior are hierarchical, and can include several different kinds of plans, as well as multiple instances of the same plan type. For example, at one level, the Plan may be *Sequential*, consisting of an ordered sequence of steps. But a given step of the sequence may be accomplished by an *Evolutionary Plan* in which the solution is obtained by retrieving a solution from the Answer Library. At yet another level of detail, the existing solution procedure found in the Answer Library would perhaps be an embodiment of some *General Principle* plan, perhaps representing a recursive solution. {Figure 2} illustrates this hierarchy.

HIERARCHICAL PLANNING

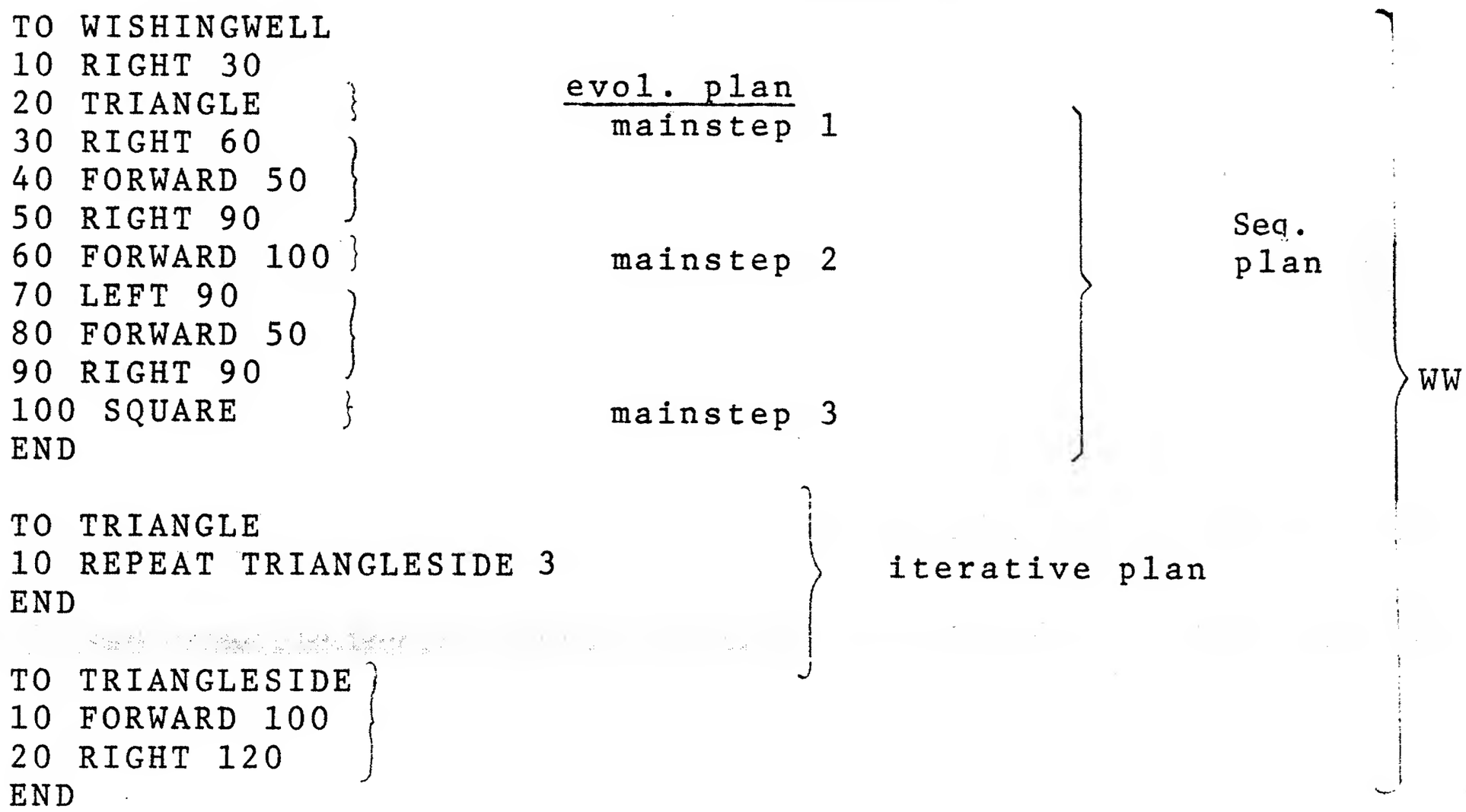


Figure 2

2. A Planning Grammar

> It would help a great deal if we had a general language specially designed for talking about Plans.... Such a language would, presumably, give us a convenient notation for such aspects as flexibility of Plans, the substitution of subplans, conditional and preparatory subplans, etc. For example, it does not particularly matter in what order Mrs. Jones chooses to run her errands when she gets to town. The ... subplans can be permuted in order, and so we say that this part of her Plan is flexible. But she cannot permute the order of these with the subplan for driving to town, or for driving home. That part of the plan is inflexible. Some subplans are executed solely for the purpose of creating the conditions under which another subplan is relevant. Such preparatory or mobilizing subplans cannot be freely moved about with respect to the other subplans that they anticipate. Another important dimension of freedom that should be analyzed is the interchangeability of subplans. Mrs. Jones can drive to town over a variety of equivalent routes. The variety is limited only by the condition that they terminate when one of her three alternative destinations is reached, since only then would the next part of her Plan become relevant. Given a satisfactory Plan and a statement of the flexibility and substitutability of its subplans, we should then be able to generate many alternative Plans that are also satisfactory. And we should like to have ways for deciding which combinations of Plans are most efficient....
>
> [Miller et al. 1960]

We view planning as a process in which the problem solver selects the appropriate plan type and then carries out the subgoals defined by that plan applied to the current problem.[4] From this viewpoint, the taxonomy of the previous section represents a decision tree of alternative plans. Following this planning process, debugging may be required. This decision process can be formalized by a context free grammar.[5] A grammar is chosen to present these rules because it provides a simple and compact representation, useful for characterizing the hierarchical structure of planning and debugging episodes. We would not argue that a context free grammar is necessarily the best formalism for representing a theory of problem solving -- in our other papers a more elaborate formalism is employed. However, we believe that the decision points implicit in the grammar correspond, for the most part, to actual choices which must be made by the problem solver at some time in the course of the solution.

The top level rule in the problem solving grammar is:

P1: SOLVE -> PLAN + (DEBUG)[6]

The nonterminal SOLVE is formally analogous to the nonterminal SENTENCE in a linguistic grammar for parsing or generating sentences.

P1 states that planning is first used to generate an almost-right Plan, with subsequent debugging then being required to complete the solution. Of course, the plan may be entirely correct, or even if incorrect, the problem solver may choose not to debug, in which case the plan remains unverified. For this reason, DEBUG is in parentheses, indicating that it is an optional constituent.

The taxonomy of the previous section characterized the planning process as involving four mutually exclusive plan categories: General Principle Plans, Evolutionary Plans, Domain Dependent Plans and Anticipatory Plans. Hence, in planning, the problem solver must choose among these alternatives. We represent this by the disjunctive rule P2.

P2: PLAN -> GP-PLAN | EV-PLAN | DD-PLAN | AN-PLAN

Now let us consider the details of each of these planning categories. Two important General Principle Plans are Sequential Plans, in which the problem is subdivided into independent parts, and Round Plans, in which the problem is characterized in terms of a sub-problem repeated some number of times.

P3: GP-PLAN -> SEQ-PLAN | RND-PLAN

Were we to include other general principle plans such as full recursion (round plans are limited to tail recursion), this rule would be extended by adding additional disjuncts.

P4 states that Round Plans can be accomplished either by iterative or recursive procedures.

P4: RND-PLAN -> ITER-PLAN | RECUR-PLAN

{Figure 3} illustrates a triangle being accomplished by three different Logo programs. These correspond to the use of a Sequential Plan, a Recursive Round Plan and an Iterative Round Plan. The annotations in parentheses, stating what the planning step is intended to accomplish, are semantic descriptions not generated by the grammar. The grammar must be supplemented by semantic interpretation rules to allow for such analysis. [Goldstein & Miller 1976b] does this.

A Sequential Plan consists of a sequence of actions, each consisting of an optional Setup Step followed by a Main Step and concluding with an optional Cleanup Step.

```
P5:  SEQ-PLAN     -> [ (SETUP) + MAINSTEP + (CLEANUP) ]*
```

The essence of a Sequential Plan is that the solutions to the Main Steps can be designed independently of each other.

For the Logo graphics programming domain, a Setup, Main Step, or Cleanup consists of either the addition of a line of Logo code or a recursive application of Solve, delimited by optional punctuation.[7]

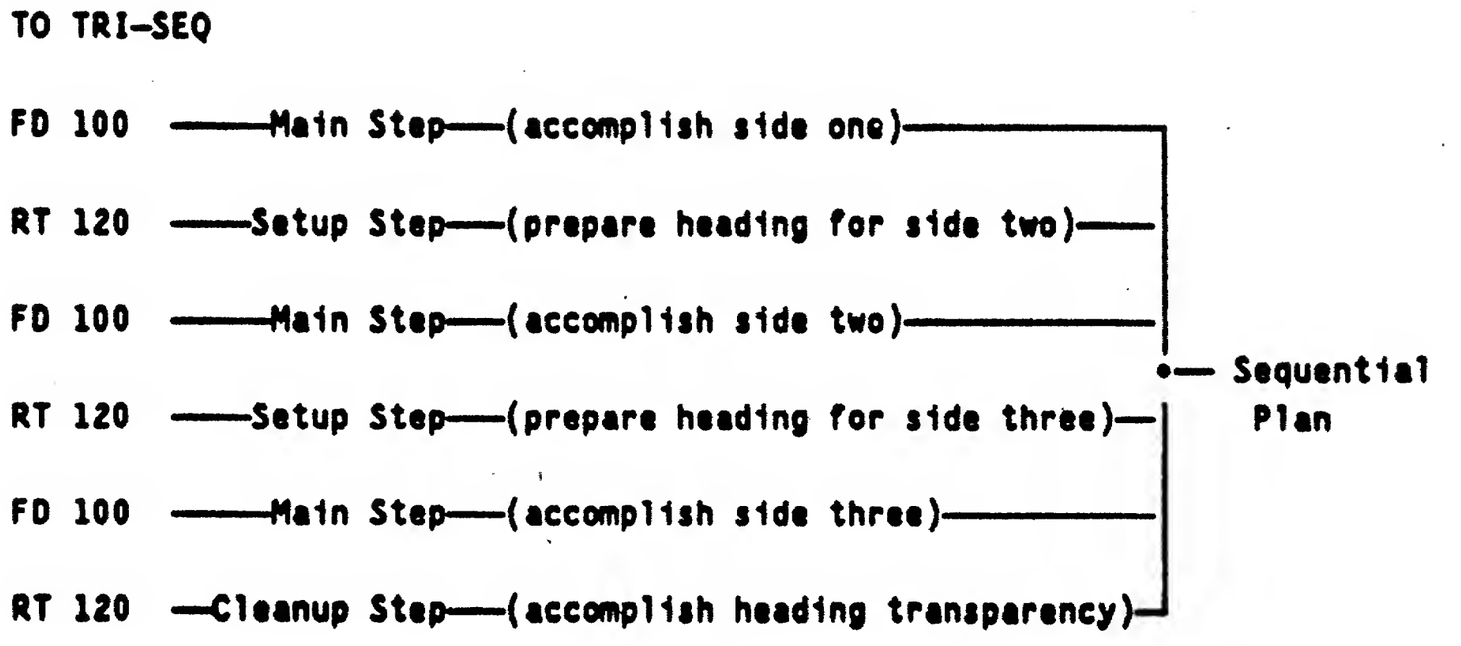
```
P6:  SETUP        -> STEP
P7:  MAINSTEP     -> STEP
P8:  CLEANUP      -> STEP
P9:  STEP         -> (PUNCT) + [ ADD | SOLVE ] + (PUNCT)
P10:  PUNCT       -> TO | EDIT | END | CS | ST | PO
```

The grammar now admits potentially infinite recursion. What is not formalized is the fact that SOLVE is always attempted with respect to some specific problem and in a definite context. Successful problem solving involves solving successively simpler problems until a direct solution in terms of the answer library is possible. The semantic component, not formalized in the current essay, would constrain the potentially infinite recursion allowed by the grammar.

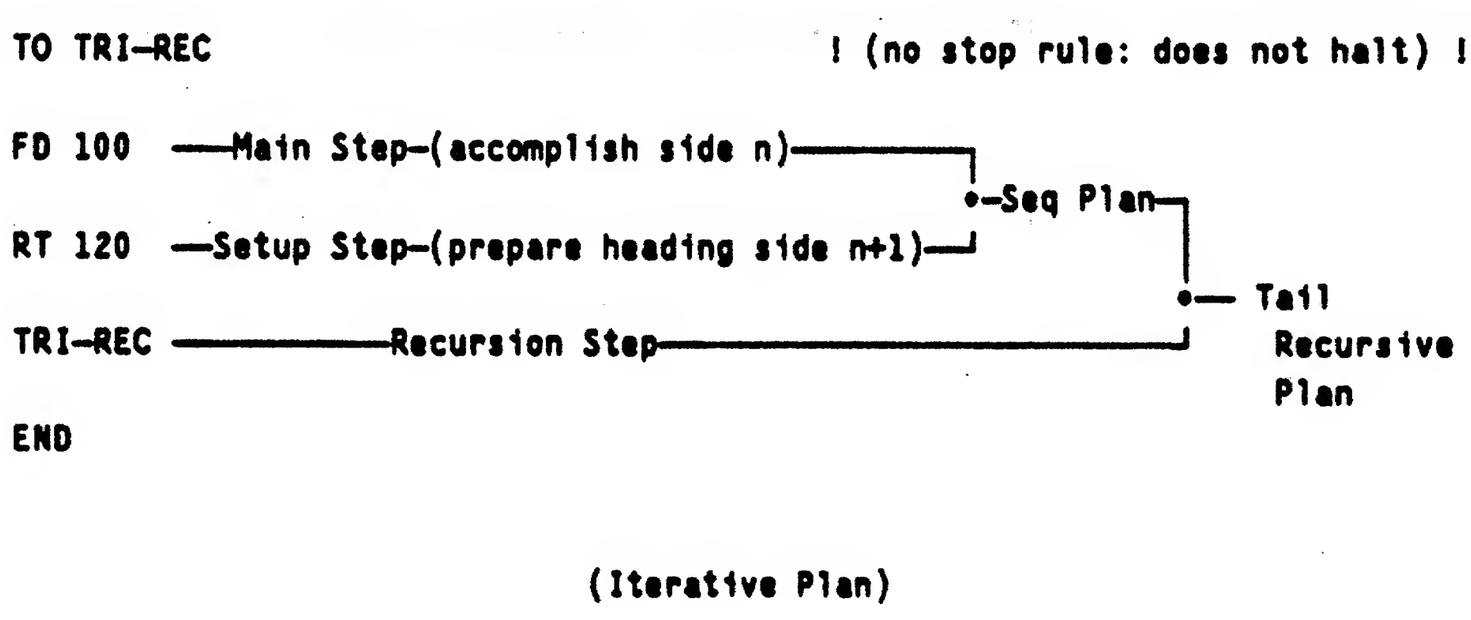
Similarly, the grammar does not capture the distinction between a SETUP, MAINSTEP, and CLEANUP: they are all simply STEPS. There is, however, a semantic distinction. For example, the distinction between a Main Step and a Setup depends on whether the code is designed to directly accomplish some subgoal (such as drawing a visible part of the picture) or to establish some prerequisite for accomplishing some subgoal (such as invisibly modifying the position or

Accomplishing A Triangle

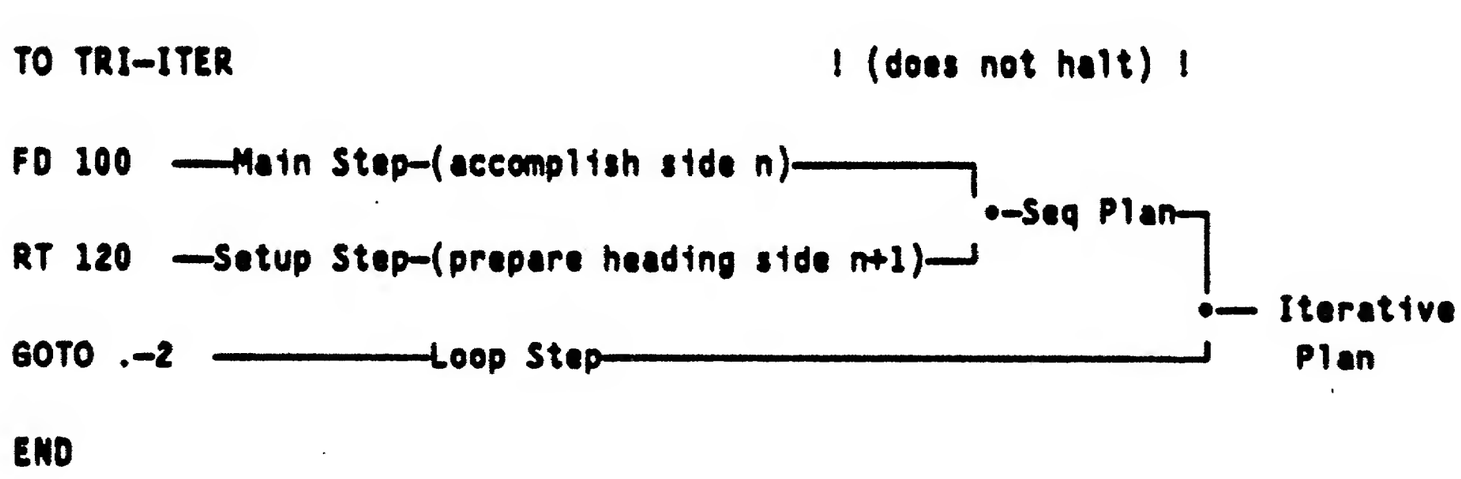
(Sequential Plan)



(Tail Recursive Plan)



(Iterative Plan)



{Figure 3}

heading of the turtle between adjacent Main Steps). The Mycroft program [Goldstein 1974] included a program annotator that made such distinctions by comparing the picture drawn by the code with the "model" (or problem description), noting the penstate, and observing the user-defined subprocedure structure. In the protocol analysis of later sections, we apply such criteria informally, when deciding which of these non-terminals to assign to a given STEP.

A simple type of Recursive Plan may be represented as a Sequential Plan plus Recursion and Stop Steps.

```
P11:  RECUR-PLAN   -> STOP-STEP + SEQ-PLAN + RECUR-STEP
P12:  RECUR-STEP   -> <RECURSIVE-PROGRAM-CALL>
P13:  STOP-STEP    -> <STOP-PROGRAM-CALL>
```

Evolutionary Plans can be based on using previous procedures, plans or models.

```
P14:  EV-PLAN      -> USE-CODE | USE-PLAN | USE-MODEL
P15:  USE-PLAN     -> <ADAPTATION-OF-PREVIOUS-PLAN>
```

For the purpose of analyzing the protocol in the next section, it suffices to assume that Evolutionary Plans based on the Use-Plan rule result in new code whose Plan is a slightly modified version of the Plan of the existing code. Further details concerning Evolutionary Plans are given in [Miller 1976].

An Anticipatory Plan modifies or experiments with a program in preparation for potential future needs. Typical instances of anticipatory strategies are generalization, documentation, and experimentation with a procedure in order to better understand its performance.

```
P16:  AN-PLAN      -> EXPERIMENT | GENERALIZE | DOCUMENT
P17:  EXPERIMENT   -> (STEP) + TRYOUT + SOLVE
```

Actual instructions to the computer consist of either primitives of the Logo language or subprocedures defined by the user. Thus, rules such as P10 and P18 bridge the gap between the nonterminal symbols of the grammar and the actual events of the student protocol which constitute the terminal symbols of the grammar.

P18: TRYOUT -> <NON-EDIT-EVENT>*

A Domain Dependent Plan for designing a step available to the student is to recall the solution directly from his or her "Answer Library."

P19: DD-PLAN -> USE-ANS-LIB
P20: USE-ANS-LIB -> <PREVIOUSLY-PARSED-CODE>

For example, after the student has learned the meaning of the FORWARD primitive, steps which simply require drawing individual vectors can be accomplished in this way. As a file of user-defined subprocedures is developed, application of P19 can result in the inclusion of any previously parsed code.

{Figure 4} presents the complete grammar. (The debugging rules are discussed in the next section.) The grammar is used for the purpose of parsing protocols. A parsed protocol, like a parsed sentence, reveals its intermediate constituents. For language, these are noun phrases and verb phrases; for problem solving they are the particular sub-plans guiding the problem decomposition.

As a generative problem solving theory, the grammar is non-deterministic. It specifies a set of possible decisions: but it does not indicate which choice should be made. The protocol provides evidence for which planning decision was made. A more complete theory would supplement the grammar with a semantics for describing particular problems situation and a pragmatics for specifying which among alternative options are appropriate for the given semantic context. This additional knowledge would extend the approach to provide a complete, deterministic theory of problem solving. Without semantics and pragmatics, one can, nonetheless, parse a protocol, revealing which decisions the problem solver has made. The constituent structure of the protocol corresponds to the sub-plans which have been pursued. But the reasons for each decision, in terms of the specific problem and pragmatic preferences, are not modelled. We take steps in this direction by moving from a context free grammar to an Augmented Transition Network in [Goldstein & Miller 1976b].

The Planning Rules alone are sufficient to describe programs as static, finished objects. Debugging Rules become necessary, however, when the analysis is extended to handle protocols in which a student is dynamically designing (or redesigning) a program. We have already briefly

A Grammar for Planning and Debugging

P1:	SOLVE	->	PLAN + (DEBUG)
P2:	PLAN	->	GP-PLAN | EV-PLAN | DD-PLAN | AN-PLAN
P3:	GP-PLAN	->	SEQ-PLAN | RND-PLAN
P4:	RND-PLAN	->	ITER-PLAN | RECUR-PLAN
P5:	SEQ-PLAN	->	[(SETUP) + MAINSTEP + (CLEANUP)]*
P6:	SETUP	->	STEP
P7:	MAINSTEP	->	STEP
P8:	CLEANUP	->	STEP
P9:	STEP	->	(PUNCT) + [ADD | SOLVE] + (PUNCT)
P10:	PUNCT	->	TO | EDIT | END | CS | ST | PO
P11:	RECUR-PLAN	->	STOP-STEP + SEQ-PLAN + RECUR-STEP
P12:	RECUR-STEP	->	<RECURSIVE-PROGRAM-CALL>
P13:	STOP-STEP	->	<STOP-PROGRAM-CALL>
P14:	EV-PLAN	->	USE-CODE | USE-PLAN | USE-MODEL
P15:	USE-PLAN	->	<ADAPTATION-OF-PREVIOUS-PLAN>
P16:	AN-PLAN	->	EXPERIMENT | GENERALIZE | DOCUMENT
P17:	EXPERIMENT	->	(STEP) + TRYOUT + SOLVE
P18:	TRYOUT	->	<NON-EDIT-EVENT>*
P19:	DD-PLAN	->	USE-ANS-LIB
P20:	USE-ANS-LIB	->	<PREVIOUSLY-PARSED-CODE>
D1:	DEBUG	->	[DIAGNOSE + (FIX)]*
D2:	DIAGNOSE	->	[DESKCHECK | TRYOUT | LOCALIZE]*
D3:	DESKCHECK	->	(PRINTOUT) + <LONG-LATENCY>
D4:	LOCALIZE	->	TRACE | PRINTOUT | ADD-PAUSE | ADD-PRINT
D5:	FIX	->	EDIT | SOLVE
D6:	EDIT	->	[ADD | DEL]*
D7:	ADD	->	<DEFINE|EDIT-EVENT>
D8:	DEL	->	<DEFINE|EDIT-EVENT>

{Figure 4}

indicated (in the third figure) the manner in which the grammar describes the planning structure of completed programs. In the next section, we describe a complementary set of debugging rules.

3. A Debugging Grammar

> ... genetic epistemology deals with both the formulation and the meaning of knowledge. We can formulate our problem in the following terms: by what means does the human mind go from a state of less sufficient knowledge to a state of higher knowledge? The decision of what is lower or less adequate knowledge, and what is higher knowledge, has of course formal and normative aspects. It is not up to psychologists to determine whether or not a certain state of knowledge is superior to another state. That decision is one for logicians or for specialists within a given realm of science. For instance, in the area of physics, it is up to physicists to decide whether or not a given theory shows some progress over another theory. Our problem, from the point of view of psychology and from the point of view of genetic epistemology, is to explain how the transition is made from a lower level of knowledge to a level that is judged to be higher. The nature of these transitions is a factual question. ...
>
> [Piaget 1971]

Planning is only the first phase in moving from problem description to solution procedure; debugging the plan is the second. A view of problem solving as alternating planning and debugging episodes is based on two theoretical observations: (a) it would be far too severe a restriction on the planner to always demand entirely correct plans; (b) debugging cannot take place in a vacuum; rather, it is a process guided by the known weak points of the particular plan that has failed.

Grammar rule D1 describes debugging as a two-stage process in which first the cause of the bug(s) (if any are discovered) is determined, and then the culpable code is either edited or redefined.

```
D1: DEBUG     -> <(DIAGNOSE) + (FIX)>*
D2: DIAGNOSE    -> <DESKCHECK | TRYOUT | LOCALIZE>*
```

D2 states three strategies for diagnosing the problem.[8] When the student prints out the developing program and then does not return to the computer for a period of time much longer than the usual intervals between typing, it may be reasonable to infer that a "desk checking" episode has occurred. D3 describes this behavior.

D3: DESKCHECK -> (PRINTOUT) + <LONG-LATENCY>

D4 specifies certain common strategies that are used to localize a bug.

D4: LOCALIZE -> TRACE | PRINTOUT | ADD-PAUSE | ADD-PRINT

The occurrence of any of these is easy to detect; however, this level of analysis is too coarse to provide much insight into the underlying reasoning. [Miller & Goldstein 1976c] and [Goldstein & Miller 1976b] provide a deeper and more detailed theory of localization, in which evidence from the plan and problem supplement the traditional examination of code (PRINTOUT) and process (TRACE, ADD-PAUSE, ADD-PRINT).

Fixing a program can be accomplished either by editing the existing code or by writing new code. The latter is described as recursively entering the grammar and solving for a program that can replace the culpable code entirely.

D5: FIX -> EDIT | SOLVE

D6, D7, and D8 terminate the debugging grammar in calls to the programming language editor.

D6: EDIT -> [ADD | DELETE]*
D7: ADD -> <EDIT-EVENT>
D8: DELETE -> <EDIT-EVENT>

Of course, the grammar is incomplete, not only with respect to the procedural knowledge necessary to apply it, either analytically to parse protocols or synthetically to predict a student's behavior; but also with respect to the set of possible plans and debugging strategies. Moreover, the grammar is both too weak and too powerful: while it fails to account for some aspects of the problem solving process which are not dealt with in this paper, it also fails to rule out some derivations which we find implausible. However, the grammar *is* adequate to analyze the structure of the protocol presented in the next sections.

4. Structural Protocol Analysis

> In assessing the validity of the program to describe or explain the subject's behavior, two things are missing to which psychologists have become accustomed. First, there is no acceptable way to quantify the degree of correspondence between the trace of the program and the protocol. This is not a problem of making the inference definite or public. Trace and protocol can be laid side by side... However, a comparison still must be made between an elaborate output statement and a free linguistic utterance. Although a human can assess each instance qualitatively, there are no available techniques for quantifying the comparison, or summarizing the results of a large set of comparisons.
>
> Second, the program has been created partly with the subject's protocol in view. Thus, something analogous to the calculation of degrees of freedom used in fitting curves with free parameters to data is appropriate. But programs are not parameterized in any simple way and no analytic framework yet exists for allowing for degrees of freedom.
>
> [Newell, On the Analysis of Human Problem Solving Protocols, 1966, pp. 3-4.]

The problem solving grammar can be utilized to analyze a protocol of a student designing, testing and debugging an elementary program. The analysis consists of constructing a *parse tree*: a structural description of the behavior corresponding to the application of appropriate grammatical rules. The parse represents a possible explanation of *one aspect*[9] of the process by which the student constructed his program. This hypothesis requires far more study than is provided in this essay. Elaborations are required along at least two dimensions. Additional theoretical assumptions are necessary in order to construct a *predictive* model of an individual's problem solving, in terms of when -- and in what fashion -- particular plans will be applied. Furthermore, the procedural knowledge employed informally to guide the parsing process must be formalized in order to automate it. Nevertheless, parsing protocols by hand with the simplified grammar presented here is a necessary starting point for a more detailed computational theory of problem solving.

Our objective, a computational model of cognition, and our methodology, the analysis of human problem solving protocols, are both closely akin to those of Newell and Simon and their colleagues at Carnegie Mellon University. (See, for example, [Newell & Simon, 1972].) The major differences are two: (1) our use of a grammar rather than a production system reflects a view of problem solving as being more hierarchical than is typical for production models (though of course

production systems can model a context free grammar); (2) while there have been careful analyses of specific problem domains such as logic and cryptarithmetic, we know of no comparable production models for planning or debugging.[10]

The protocol we shall examine is of a student named Greg designing a program to draw his name, a typical introductory Logo project.[11] The protocol was obtained, and is presented, with the student's permission. Greg was a high school freshman with four previous Logo sessions, each approximately ninety minutes in length. It should be mentioned that the investigation of the student's planning and debugging was not unobtrusive. He was being taught many of the strategies discussed in this document.

{Figures 5} shows an initial abridged segment of the protocol, in which Greg is writing a subprocedure to draw the letter G. The several hundred lines of the total protocol, indeed even the far fewer events shown in the G protocol, are by themselves virtually incomprehensible. {Figures 6} presents a parsed version of the G protocol which we believe to be a far more comprehensible description. This description was generated by hand, but using relatively clearcut guidelines described in the next section.[12]

Let us consider what aspects of the student's problem solving behavior the parse reveals:

1. The parse indicates the kind of plan Greg applied to the problem. For the G, Greg used a sequential plan. Later we examine Greg's protocol for the R and infer that he applied a combination of evolutionary and anticipatory planning strategies.

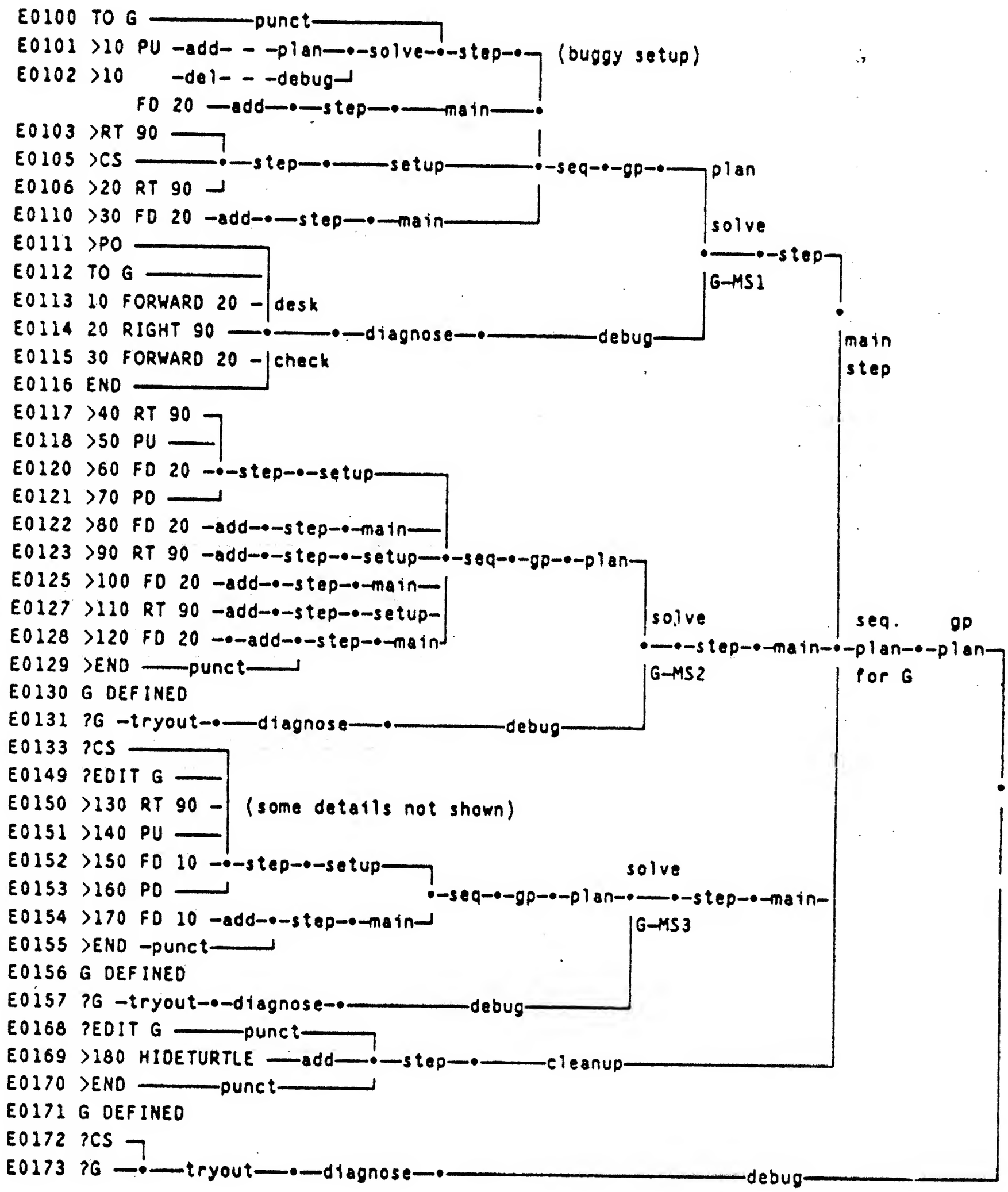
2. For the particular kind of plan, the parse indicates the subgoal structure Greg chose. Without evidence from the protocol, a human programmer or an automatic program understanding system might expect a subgoal structure for a G of the kind shown in {Figure 7}. But the structure of Greg's debugged G program illustrated in {Figure 8} shows another decomposition. If one takes advantage of the protocol clues indicating where Greg paused in defining the G program to do various diagnoses and repair, then a decomposition even more in keeping with how he dealt with the problem becomes apparent. {Figure 9} shows the modular structure of the program based on the full protocol. This parse for the finished program is called the "collapsed plan" and is arrived at in an algorithmic way by a simple pruning procedure on the

Unparsed GREG Protocol, Part One

```
E0100 TO G
E0101 >10 PU
E0102 >10 FD 20
E0103 >RT 90
E0105 >CS
E0106 >20 RT 90
E0110 >30 FD 20
E0111 >PO
E0112 TO G
E0113 10 FORWARD 20
E0114 20 RIGHT 90
E0115 30 FORWARD 20
E0116 END
E0117 >40 RT 90
E0118 >50 PU
E0120 >60 FD 20
E0121 >70 PD
E0122 >80 FD 20
E0123 >90 RT 90
E0125 >100 FD 20
E0127 >110 RT 90
E0128 >120 FD 20
E0129 >END
E0130 G DEFINED
E0131 ?G
E0133 ?CS
E0149 ?EDIT G
E0150 >130 RT 90
E0151 >140 PU
E0152 >150 FD 10
E0153 >160 PD
E0154 >170 FD 10
E0155 >END
E0156 G DEFINED
E0157 ?G
E0168 ?EDIT G
E0169 >180 HIDETURTLE
E0170 >END
E0171 G DEFINED
E0172 ?CS
E0173 ?G
```

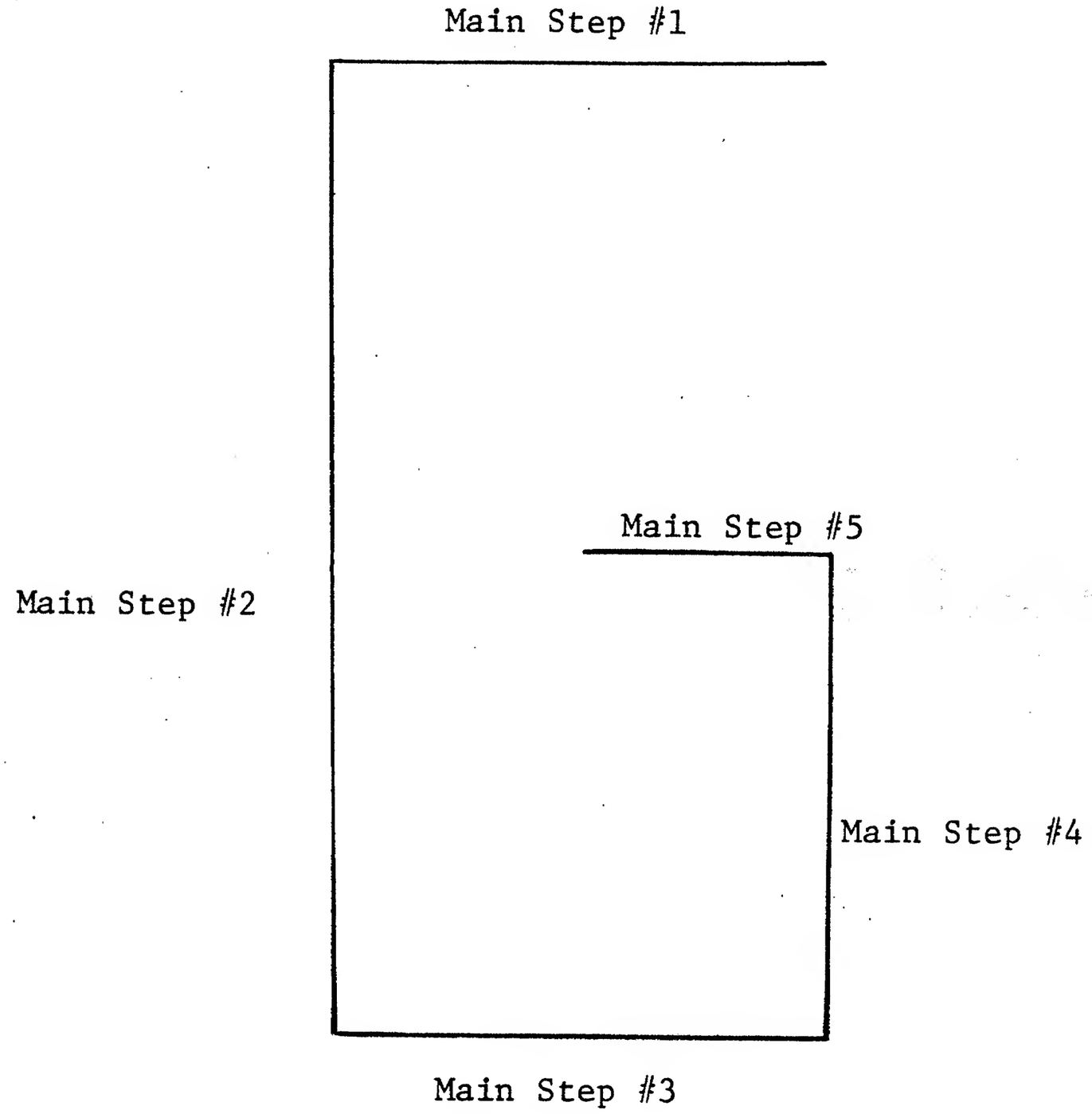
{Figure 5}

Parsed GREG Protocol, Part One



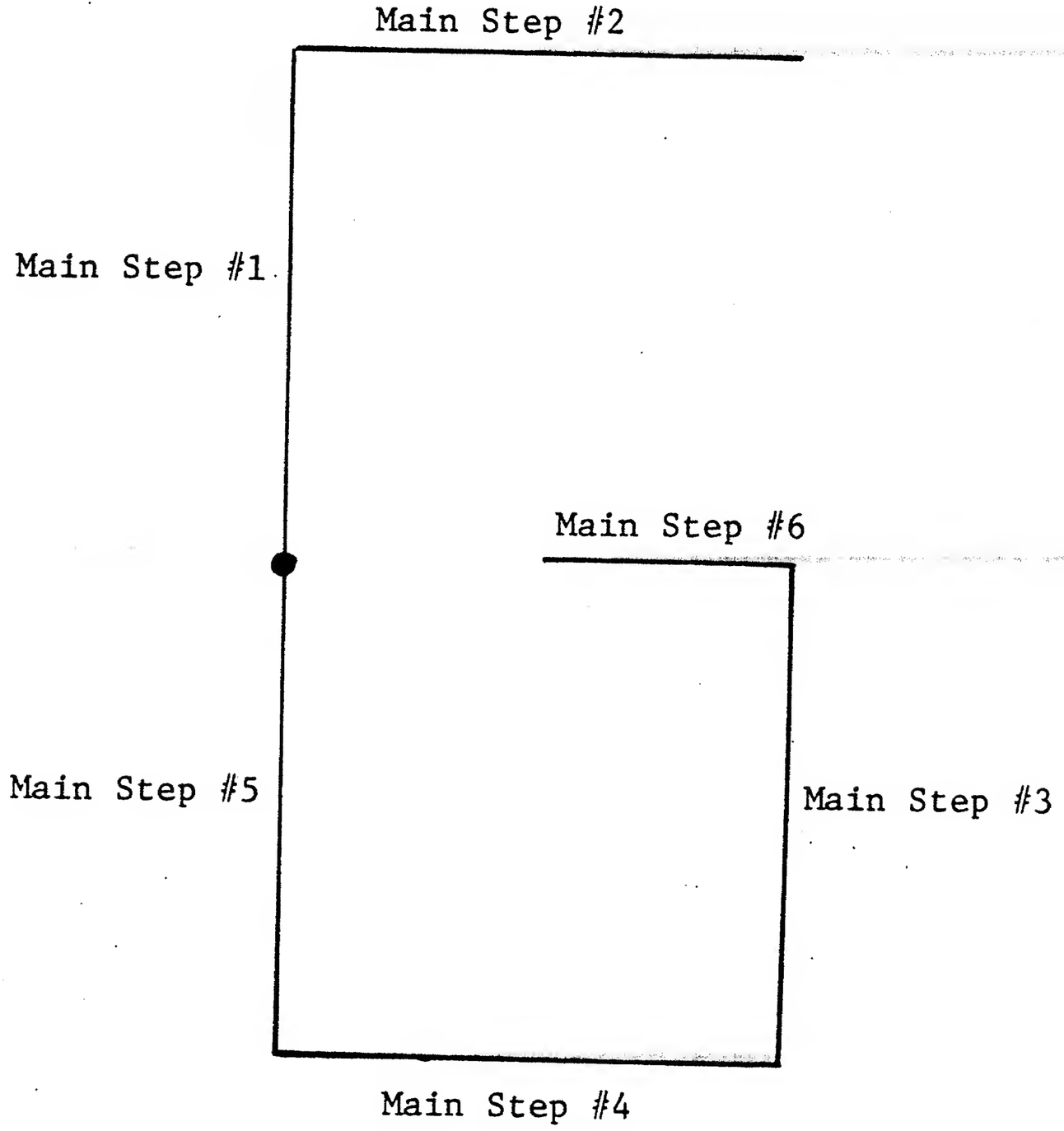
{Figure 6}

Expected Sub-goal Structure for a G



{Figure 7}

Apparent Sub-goal Structure from Finished G Code



{Figure 8}

A Segment of Protocol, and its "Collapsed" Plan

The Segment of Protocol

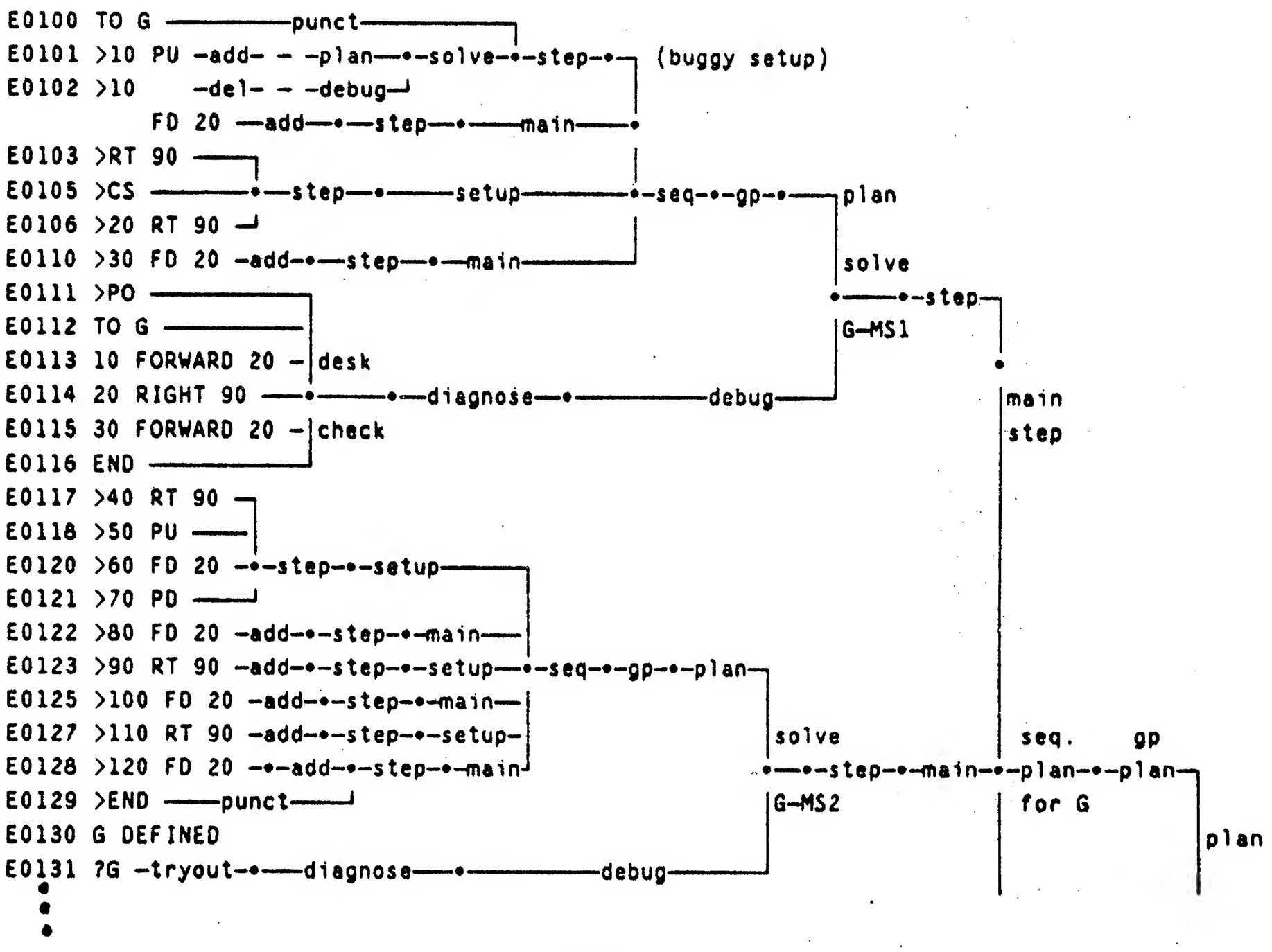


Figure 9a

Most Probable Actual Sub-goal Structure Using Protocol Clues

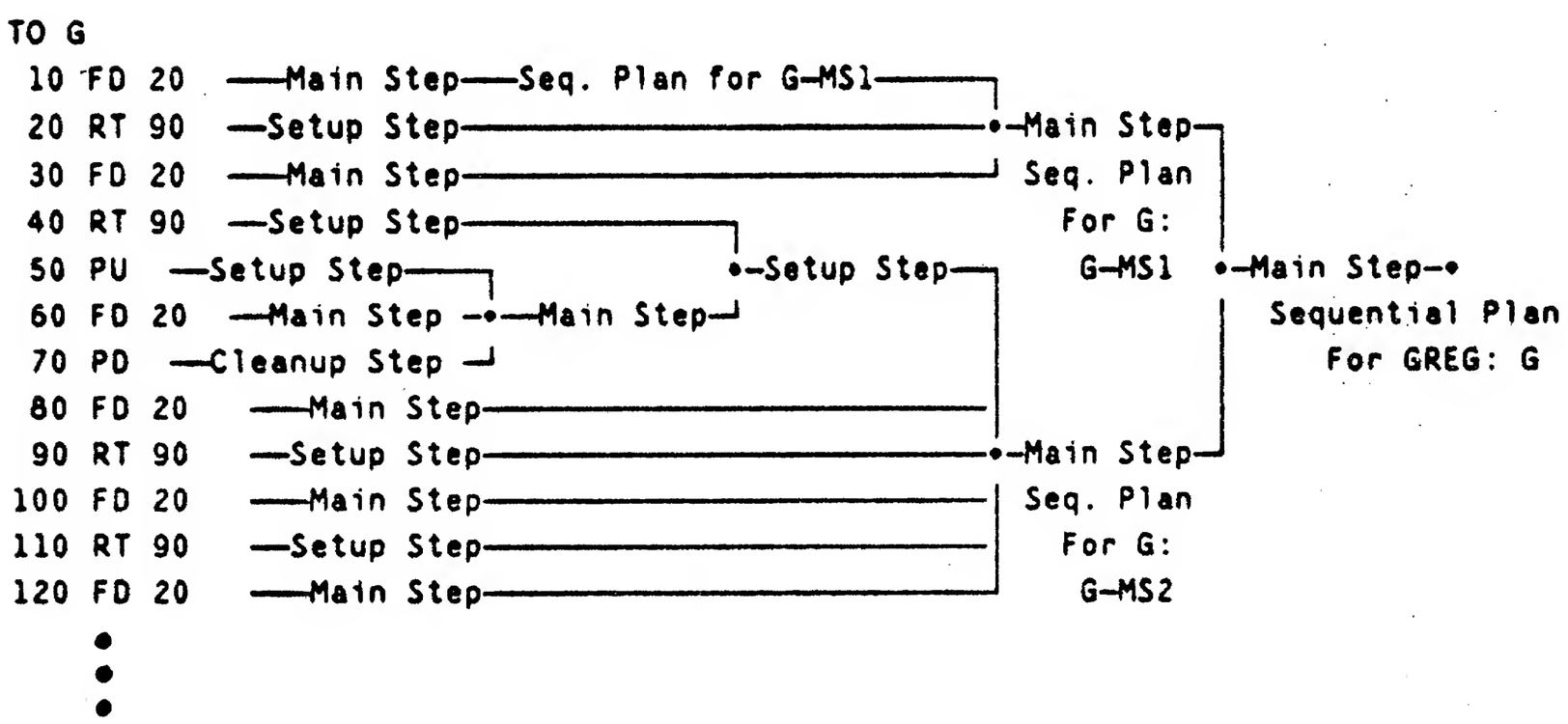


Figure 9b

complete parse as follows:

> All events in the protocol that are not part of the final program definition are deleted. Then all nonterminals that become singleton nodes in the abridged parse tree are recursively eliminated. Thus only nonterminals of the planning grammar involved in an explanation of the final program remain. Debugging operations are deleted since either the events that they point to have been abridged and are not part of the finished program, or, due to previous deletions, the debugging operations have become singleton nonterminal nodes.

Did Greg view the problem in exactly this way? Probably not in every detail. But the parse allows us as psychologists or teachers to begin a deeper study of Greg's problem solving with the evidence of the protocol clearly delineated. In future work, we envision interviewing and careful observation to understand more precisely the correspondence between the parse and the student's perception of his own problem solving.

3. Examination of this and other parsed protocols for Greg reveals the existence of certain *cliches* [Solomon 1976] in his problem solving. By a cliche, we mean a repeated piece of sub-structure called forth by certain circumstances, regardless of its appropriateness. For example, Greg had a clear tendency to begin every procedure with a PENUP (as in Event 101). In terms of our grammar, we would say that for Sequential Plans, whose grammar rule is:

P5: SEQ-PLAN -> [(SETUP) + MAINSTEP + (CLEANUP)]*

Greg did not treat the initial setup as optional. Rather, his plans seemed to reflect a modified rule:

P5': SEQ-PLAN -> [SETUP + MAINSTEP + (CLEANUP)]*

Another cliche present in this parse whose frequency was apparent in examining other protocols was Greg's tendency to use the CLEARSCREEN command as a transition from planning to debugging. This is not required, when the debugging involves simply changing a definition. But whenever the bug involved

an unwanted line on the screen, Greg cleared the screen before correcting the definition.

4. The parse reveals how often Greg needed to debug particular kinds of plans. Greg, for example, had more difficulty with domain dependent plans (such as piecewise linear approximation of a circle) than general principle plans (such as sequential or round plans in general).

5. An examination of this and other parsed protocols reveals which planning methods Greg was familiar with and which he never used. While sequential and round plans were in Greg's repertoire of problem solving techniques at this point in his education, a simplified kind of *interrupt plan*, in which the solution to one sub-problem is made state transparent and then inserted in the midst of another, never occurred.

We do not claim that an insightful teacher carefully examining Greg's problem solving behavior could not reach similar conclusions. Indeed, if our formal analysis is useful, it *should* provide similar observations to those of a tutor. The more formal grammatical approach has the virtue of (a) making explicit the problem solving knowledge that the teacher may only know intuitively, thereby facilitating more uniform and articulate analyses[13] and (b) moving us closer to the capability for automating protocol analysis, thereby freeing teacher time for interaction with the student.

Let us look more closely at the possibility of modelling the problem solving knowledge of the individual with reference to a formal grammatical theory of planning and debugging. The grammar we presented earlier in the paper was an "expert grammar" in the sense that no incorrect or incomplete plans were present. Let us call this the *archetypal grammar*. Our perspective on modelling is based on perturbing this archetype to reflect the knowledge of a particular student. A *personal grammar* could be constructed from the archetype using the following guidelines:

1. Ignorance of certain strategies, as evidenced by their continual non-appearance in parses (even for situations wherein the strategy is deemed highly applicable), would result in deleting those strategies from the disjunctive rule in which they appear. Thus, for Greg, at this point in his education, he is apparently unaware of plans involving full recursion. Hence, the rule for GP-PLANS need not include this possibility.

2. A tendency to always err by including an optional constituent such as an initial Setup, even where unnecessary, can be modelled by modifying the grammatical rule in which this constituent appears so that it is no longer optional. Greg, for example, had a tendency to always include a PENUP command as a setup in his sequential plans.

3. Similarly, the complementary tendency to never include an optional constituent can be modelled by deleting the constituent from its rule. Most beginners when they first learn recursion in Logo do not bother with the STOP-STEP. (The graphic picture produced is one in which the turtle moves endlessly around some figure.)

The validity of a given perturbed grammar as a model of an individual can be judged by its relative simplicity, and tested by the extent to which it successfully parses protocols produced by the individual. Of course, a student learns and we expect the grammar model for the individual to change over time. But over the short term there will be a certain constancy. Over the long term, the changes in the grammar constitute a model of the knowledge being learned by the student.

In this paper, we do not explore further the construction of personal grammars as individual cognitive models. Instead, the next section explains how we arrived at the parse for Greg's protocol. This begins a process that we hope will put us in a position to explore further the construction of these personal grammars. If ultimately successful, we will have constructed a formalism for describing and analyzing individual cognitive differences: an important step beyond the statistical notion of educational evaluation that is now prevalent.

5. Analysis of a Sequential Plan for Drawing a G

This section begins the discussion of how we arrived at the parse of Greg's protocol. Here we analyze the first part of the protocol in which he constructs a program for drawing the G. In the next section, we examine an evolutionary sequence for achieving the R.

In deriving the parse, we applied the following criteria:

1. The grammar constrains the possible parses. Indeed, the grammar is already personalized somewhat in that we did not include in the planning rules a number of plans unfamiliar to Greg, such as full recursion.

2. Which plan is being used is determined by bottom up evidence related to the control structure: round plans involve iteration or tail recursion; sequential plans involve in-line coding or sequences of subroutines.

3. As teachers we had already acquired some insight into Greg's typical cliches by the time this fourth session was analyzed; we preferred parses consistent with our previous generalizations to alternatives which would have required additional assumptions.

4. As programmers, we had some expectations about how a project such as drawing a G would be accomplished. These expectations had to be supported by the code, but they nevertheless directed our attention to consider certain parses before others.

A standard issue in parsing is whether to work from the top downward, or from the terminal units upward. An automatic protocol analyzer for dynamic use in tutoring would be further constrained to proceed in left to right order (i.e., forward in time) as well. Backing up could result in inappropriate tutorial commentary, or missed opportunities for providing assistance. The presentation here is not systematic with respect to either dimension. There are clear cases where lookahead could not be avoided; some of these are pointed out. Our goal here is to argue for the viability of our theory as the basis for further research. We intend to implement the associated parsing programs, thereby resolving some of these issues.

The parse represents our interpretation that Greg's top level approach to generating a GREG procedure was a Sequential Plan. Recall that our definition of a Sequential Plan is one in which the problem is divided into subgoals, each solved independently and then recombined in sequential order. In this case, the problem of drawing the word GREG was divided into the independent subgoals of achieving each of the letters. The recombination to accomplish the overall goal was done via the usual sequencing of text in left-to-right order.

Note that there is a choice in the application of Sequential Plans. The top level procedure could be defined, followed by the recursive expansion of lower levels; this amounts to a *breadth first* analysis. Greg pursued the alternative *depth first* strategy, taking the first main step and immediately expanding that. Thus, his initial goal was to define the "G" program. The superprocedure for the entire name is not constructed until the latter half of the session. (This is a decision which must be made by the problem solver which is not apparent in the grammar rules. A theme of our current research is to refine the formalism so that all such decision points become explicit.)

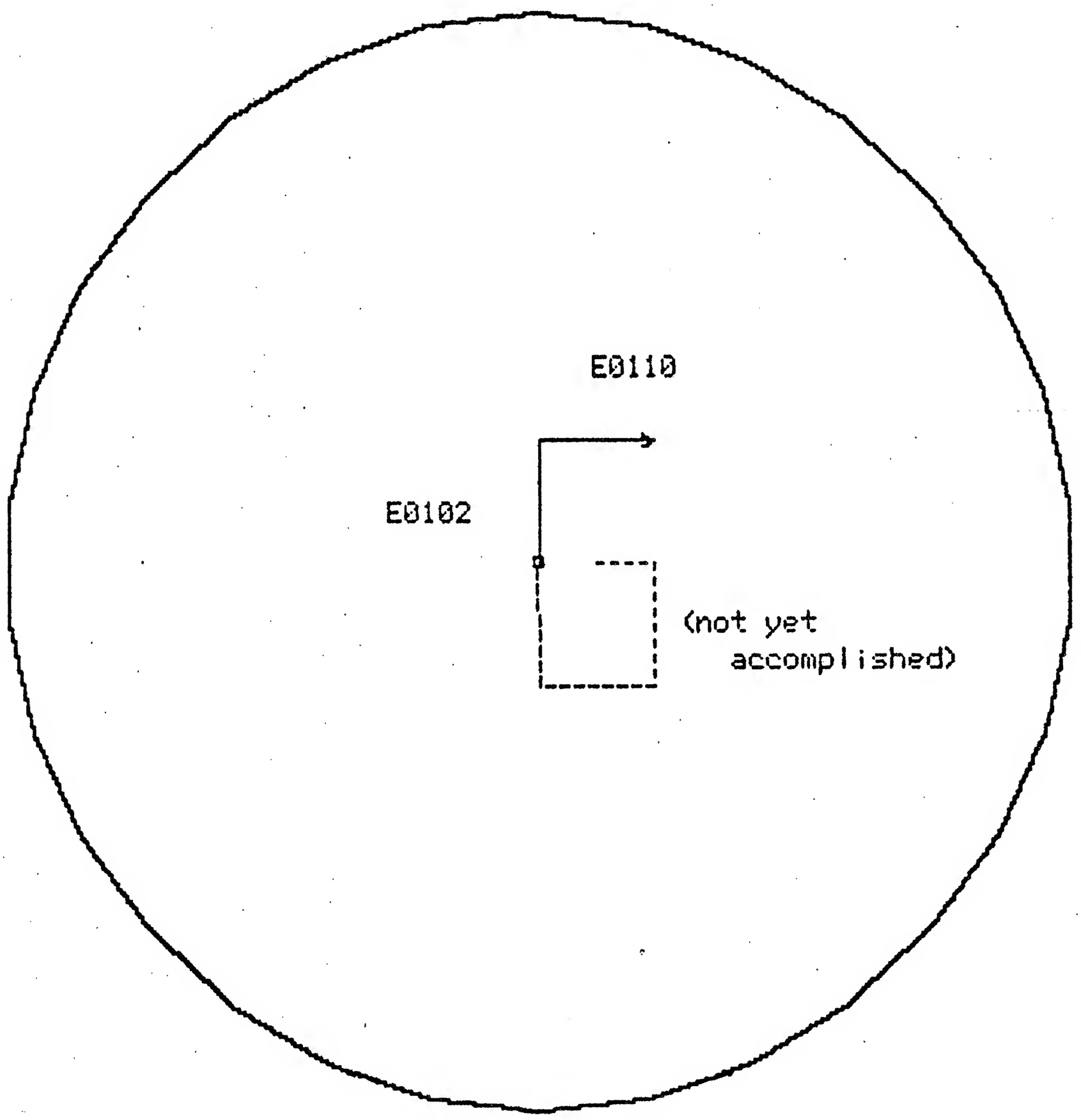
The G was also accomplished by a Sequential Plan. Events E0100 through E0110 constitute the first episode, which is the definition of a code segment for the initial part of the letter G. See {Figure 10}.

```
E0100 TO G
E0101 >10 PU
E0102 >10 FD 20
E0103 >RT 90
E0105 >CS
E0106 >20 RT 90
E0110 >30 FD 20
```

Event E0101 is a PENUP. Such instructions at the beginning of a procedure usually signify a Setup Step in which the turtle is being moved to the initial position for the first Main Step. But since the turtle is already at the initial position for G, the PENUP is unnecessary. Event E0102 indicates that the student has recognized this and corrected it by typing another Logo line with the same line number. This has the effect of deleting the PENUP of E0101. An alternative interpretation is that the student is designing a position Setup and that E0101 and E0102 were intended to have distinct line numbers. But this interpretation is less probable, since, in fact, the G

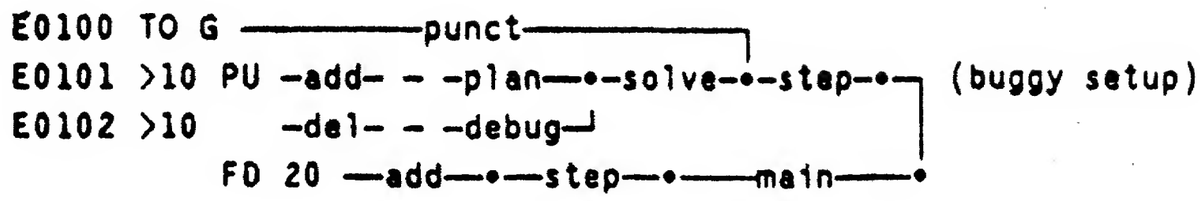
To Draw a G, version 1

(dotted lines show future additions)



{Figure 10}

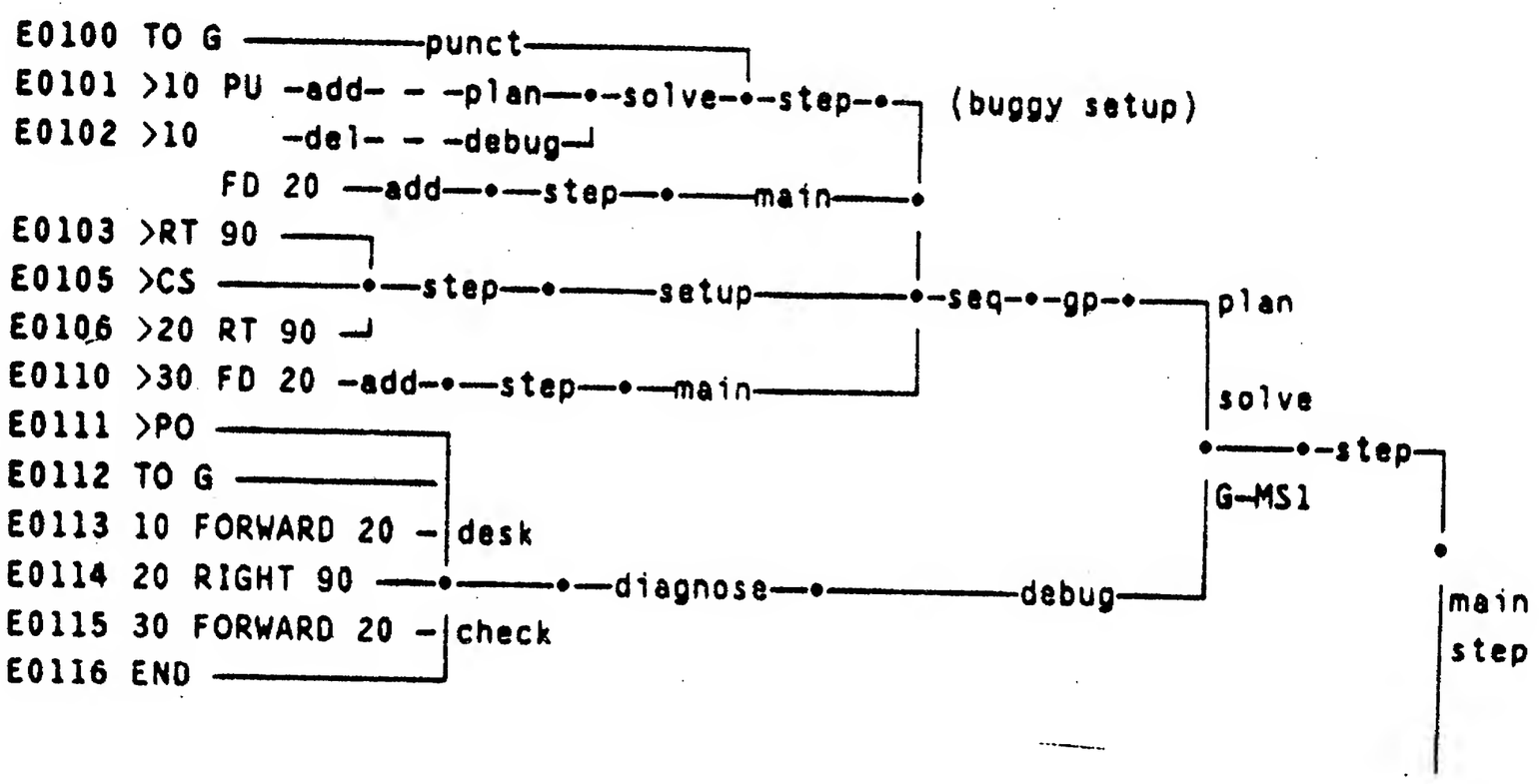
does not require such a preparatory step.



Events E0103 through E0106 are another subgrouping, in which an attempt to type in a Setup Step for the second stroke of the G fails. The bug is the omission of line number 20 in E0103. This error is so frequent that it will not be mentioned again in the analysis. Event E0105 is interesting in that it represents a common recovery behavior -- the screen is invariably cleared [CS] after errors -- thus it can serve as a useful clue for an automated parser.

Event E0106 corrects Event E0103 by typing the line over *with* the line number.

Events E0111 through E0116 were a printout of the partly defined procedure. In the parsed figure, they are interpreted as an instance of desk checking the partially completed code. Such debugging events segment the protocol such that at an intermediate level of the parse tree is a Sequential Plan for the G whose Main Steps (e.g., G-MS1 in the figure) are "chunks" rather than single vectors; each is in turn expanded by a Sequential Plan whose Main Steps are individual vectors.



Of course, it is not possible to be absolutely certain that the student's overt behavior is an accurate reflection of his covert thought processes. The annotation is an educated guess based upon certain empirical evidence. For example: how can the interpretation of a PRINTOUT [PO] as a debugging event, rather than as punctuation, be justified? Sometimes the student explains his intention when printing out the code, and there is no reason to disbelieve him. Sometimes the PO is followed by testing and then editing, thereby increasing the likelihood that the PO was the desk checking part of a debugging episode. There are occasions when a different interpretation of PO is preferable, however, such as when the student performs PO ALL and then goes home for the day. Since alternative interpretations do not seem appropriate here, and since the interpretation given results in an expected division of the protocol into planning and then debugging sequences, the desk checking explanation is accepted. The absence of further testing and patching can be accounted for by assuming that the desk check concluded that the procedure was correct so far.

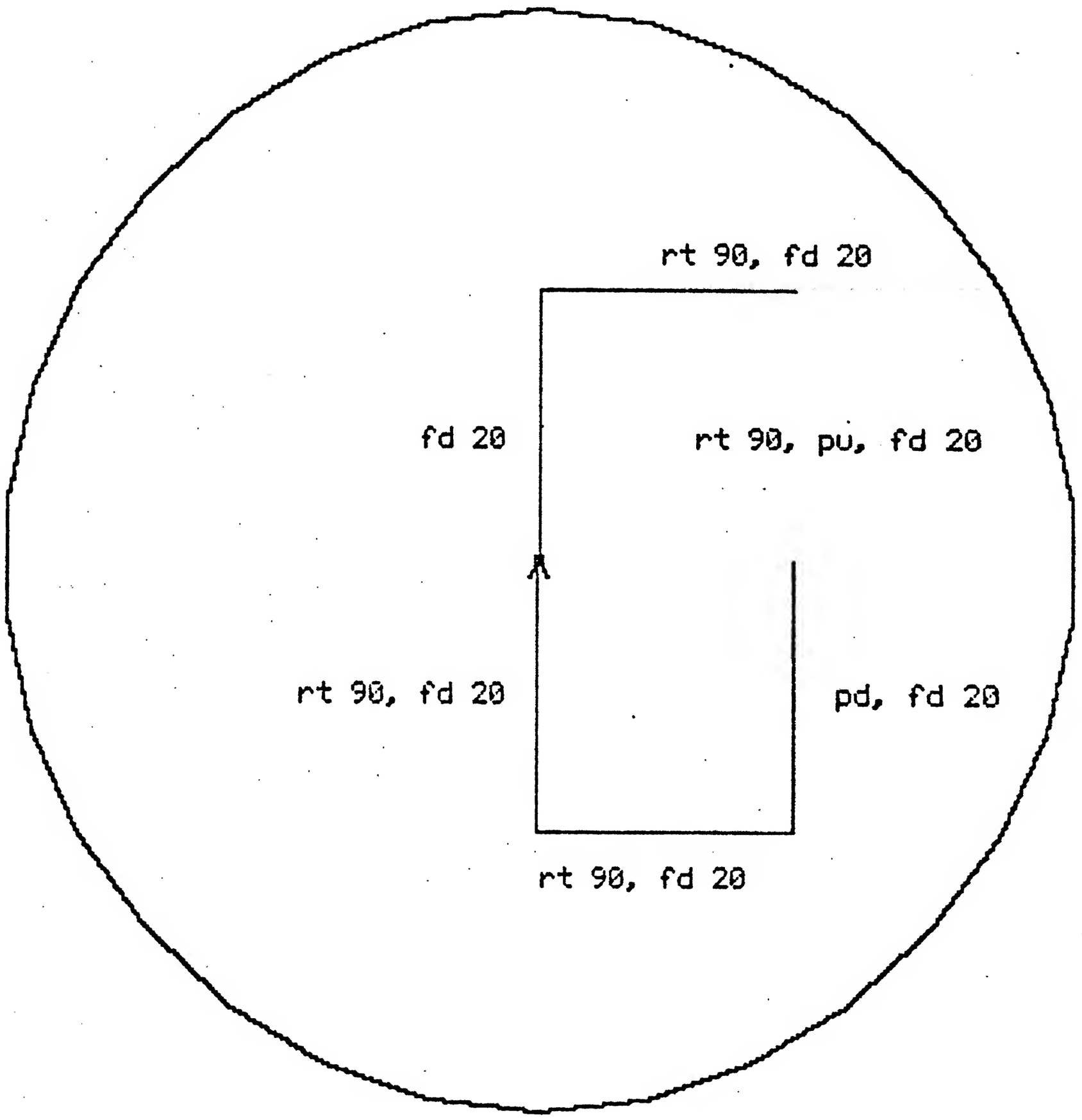
For brevity, the next part of the protocol (E0117 to E0130) is not presented in detail. Greg continued defining the G procedure. The episode is still within the context of a Sequential Plan. We resume the discussion at Event E0131, a Tryout of the G program.

E0131 ?G (See (Figure 11).)
E0133 ?CS

E0131 is successful in drawing the first five visible vectors of the G. Event E0133, a

To Draw a C, version 2

(As of E0131)



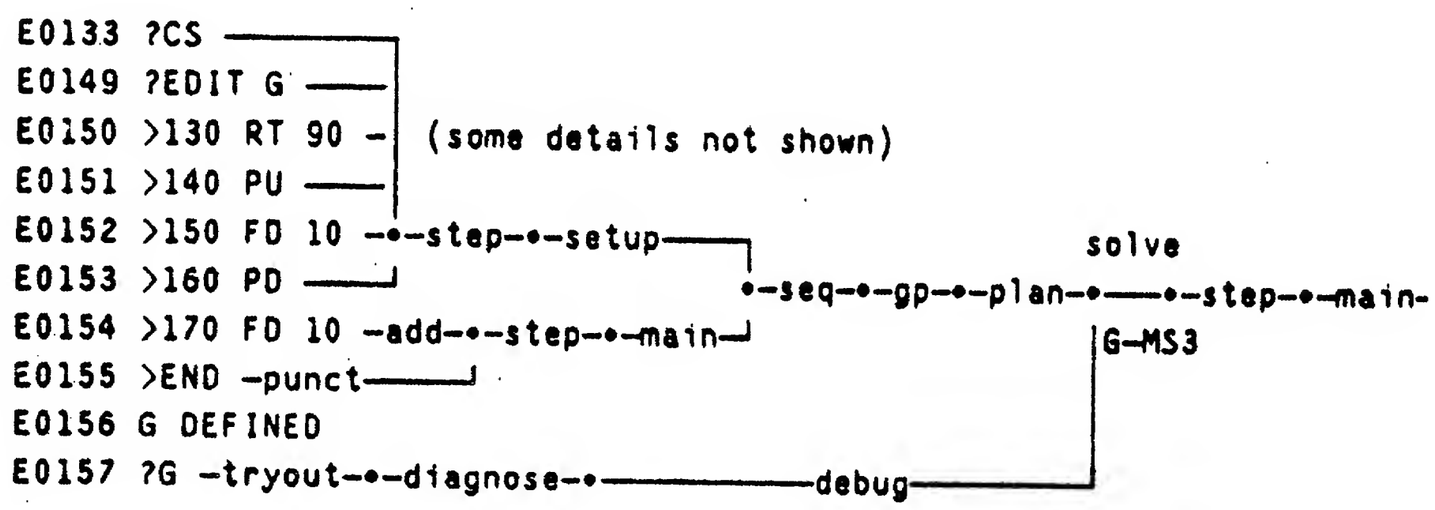
{Figure 11}

CLEARSCREEN [CS], is a frequent signal to expect a mode change. Execution of a CS which serves no preparatory function (as is the case here, since a clear screen is not a pre-requisite for defining procedures), and which does not eradicate an error, provides strong evidence that a segment boundary may exist at this point in the protocol. Here, it flags the end of testing and the commencement of planning the next step. (Events E0134 through E0148, deleted, were a PRINTOUT of the G procedure as currently defined.)

The next episode (E0149 - E0156) is the addition of code preparing for and accomplishing the last Main Step of the G.

```
E0149 ?EDIT G
E0150 >130 RT 90
E0151 >140 PU
E0152 >150 FD 10
E0153 >160 PD
E0154 >170 FD 10
E0155 >END
E0156 G DEFINED
E0157 ?G                (See (Figure 12).)
```

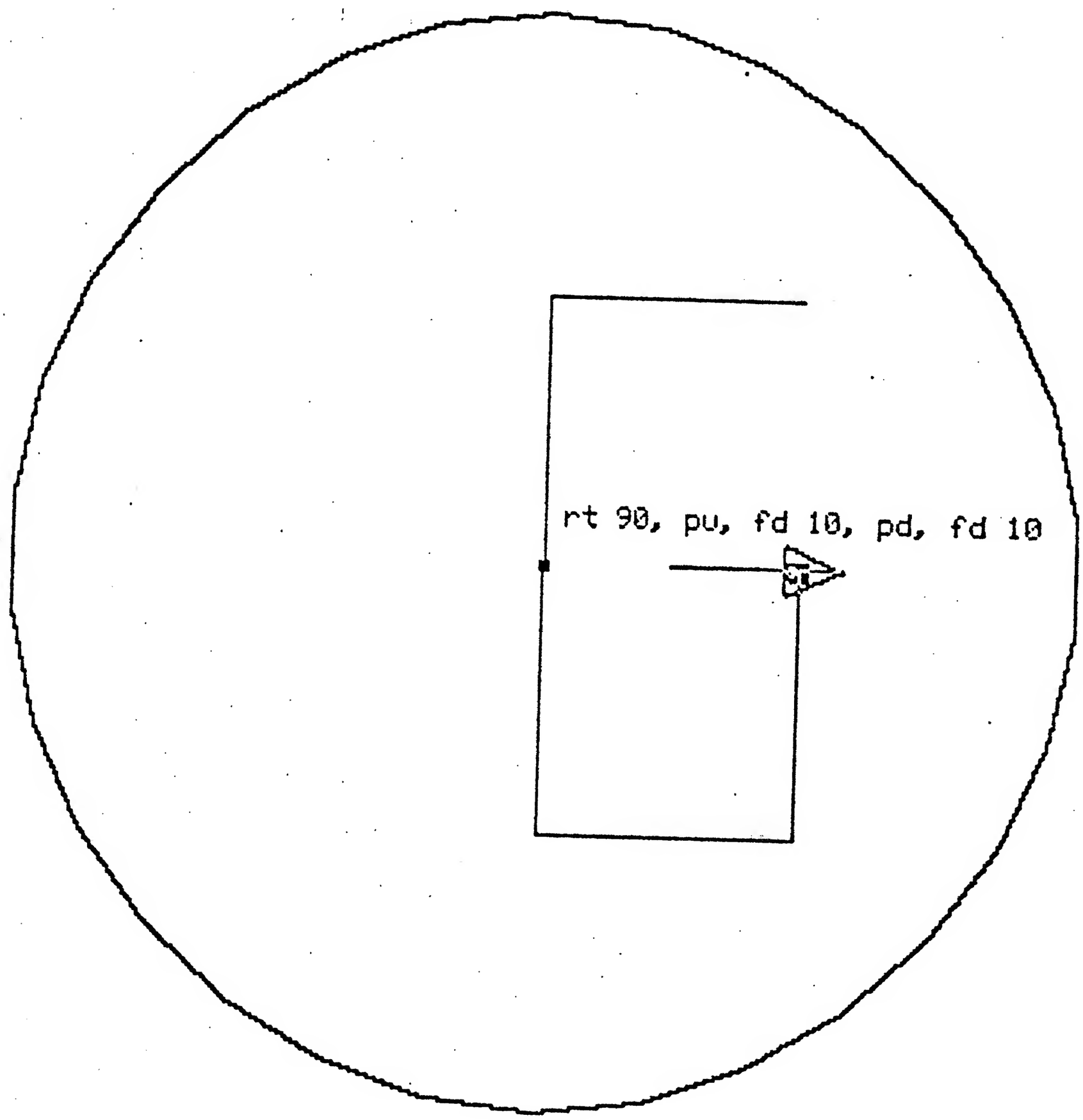
All of the Main Steps of G have now been defined. Greg verifies this by executing the program (E0157).



The final part of a Sequential Plan is the design of an optional Cleanup Step. In this case, the turtle is still appearing on the screen. A Cleanup Step is defined to eliminate this.

To Draw a C, version 3

(As of E0157)



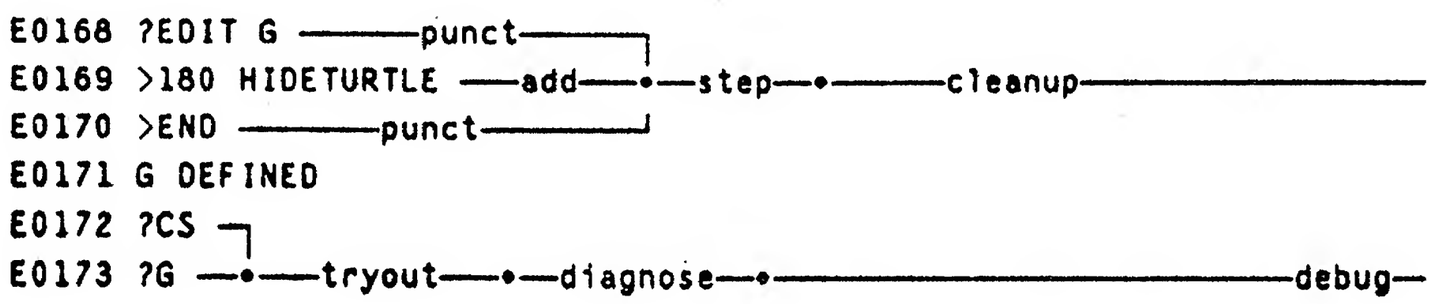
{Figure 12}

```
E0168 ?EDIT G
E0169 >180 HIDETURTLE
E0170 >END
E0171 G DEFINED
```

A final Tryout, signalled by the CLEARSCREEN of E0172, is done to verify that the G procedure is satisfactory.

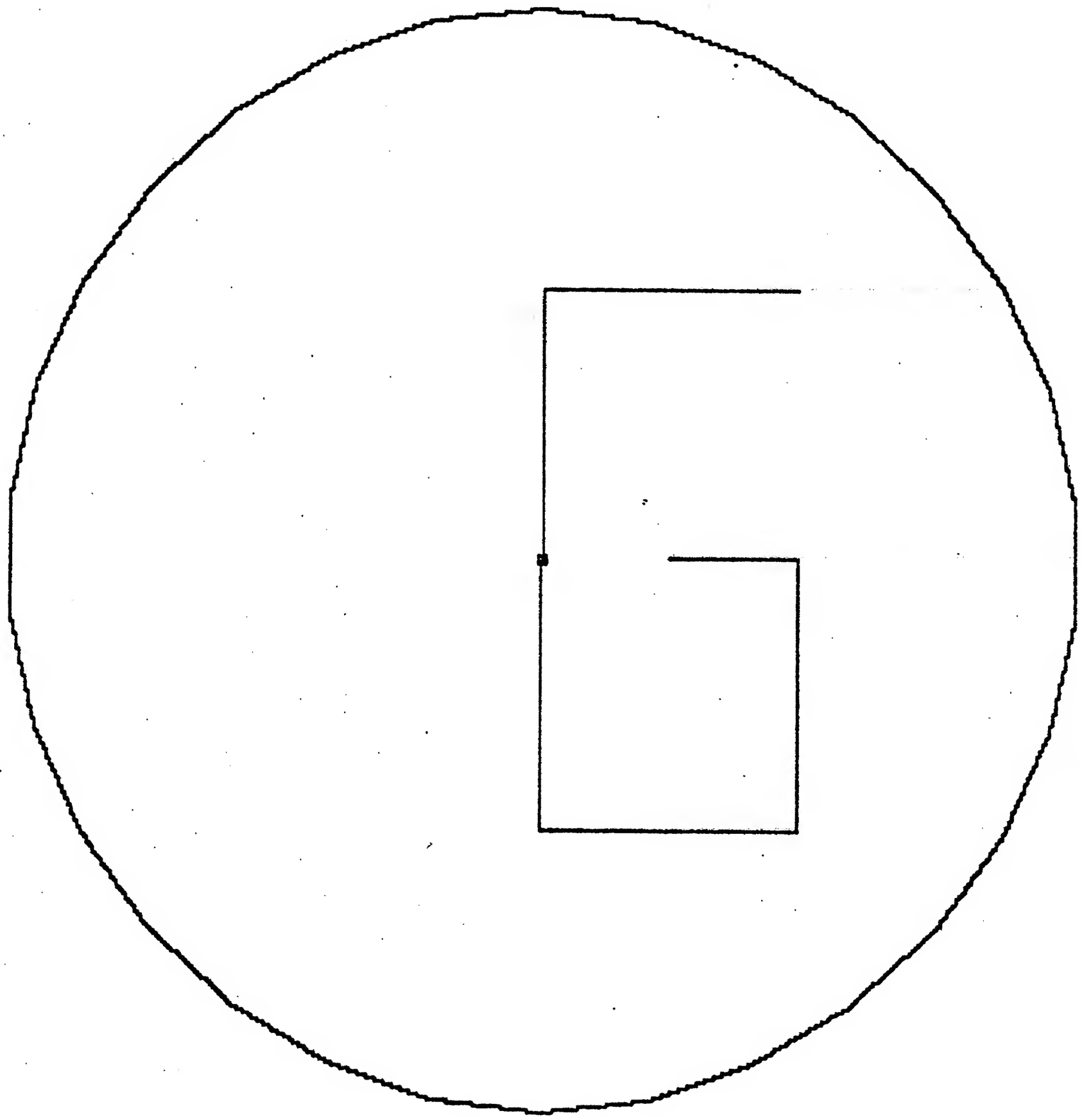
```
E0172 ?CS
E0173 ?G          (See (Figure 13).)
```

G now satisfies its specifications, and Greg resumes the higher level Sequential Plan currently in effect for accomplishing his entire name. (The SHOWTURTLE command of E0174 is further evidence of this segment boundary.) The next Main Step in this plan is to design the R program.



To Draw a C, finished

(E0173)



{Figure 13

6. Analysis of an Evolutionary Sequence for Drawing an R

{Figure 14} shows the segment of protocol during which Greg is constructing his R program. {Figure 15} shows the parsed version. The interesting part of this protocol involves Greg's exploration of a CIRCLE program. He requires a semi-circle for the R. Previously he had been exposed to the standard program for drawing circles in Logo. His behavior involves experimenting with that program, modifying it until it serves his purpose in the R. We do not show all of that here. The parse does show, however, an Anticipatory Plan in which the full circle procedure is examined, in expectation of its use in the R semicircle. The fine structure of Greg's experimentation is evolutionary, involving successive modifications of the circle.

We now begin a detailed analysis of how the parse of the R session was derived. Event E0174 commences the definition of the Main Step for R in the GREG plan.

```
E0174 ?SHOWTURTLE
E0176 ?TO R
E0177 >10 PU
E0178 >20 FD 10
```

{Figure 16} shows what would be drawn by the first two lines of the R procedure, in relation to the R as eventually completed.

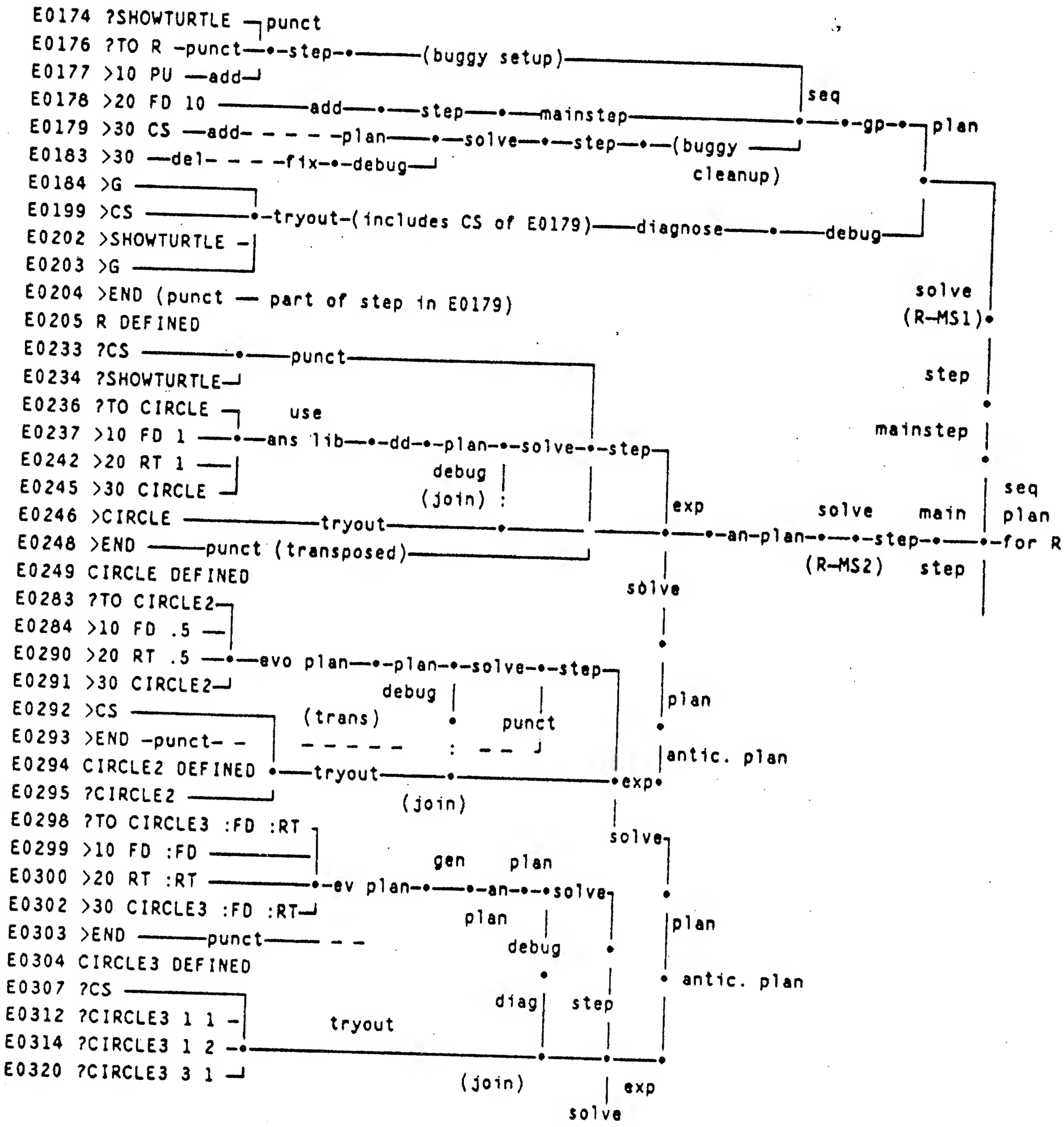
Very careful analysis is required in parsing student protocols. Events E0177 and E0178 provide a good example of the subtleties which can be involved. The PENUP of E0177 would normally indicate preparation for a position Setup. But the relation of the FD 10 in E0178 to the finished R implies that it is intended to draw a visible first part of the R, not merely an invisible position Setup for the second part. It is possible but unlikely that Greg intends this as a Setup and Plans to redraw the vector with the pen down later. Greg's predilection for ordered Sequential Plans, in which the first part to be executed is coded first, also argues that the FD 10 (Event E0178) is intended to draw a visible vector. Consequently the preceding PENUP of E0177 is interpreted as a bug.

Unparsed GREG Protocol, Part Two

```
E0174 ?SHOWTURTLE
E0176 ?TO R
E0177 >10 PU
E0178 >20 FD 10
E0179 >30 CS
E0183 >30
E0184 >G
E0199 >CS
E0202 >SHOWTURTLE
E0203 >G
E0204 >END
E0205 R DEFINED
E0233 ?CS
E0234 ?SHOWTURTLE
E0236 ?TO CIRCLE
E0237 >10 FD 1
E0242 >20 RT 1
E0245 >30 CIRCLE
E0246 >CIRCLE
E0248 >END
E0249 CIRCLE DEFINED
E0283 ?TO CIRCLE2
E0284 >10 FD .5
E0290 >20 RT .5
E0291 >30 CIRCLE2
E0292 >CS
E0293 >END
E0294 CIRCLE2 DEFINED
E0295 ?CIRCLE2
E0298 ?TO CIRCLE3 :FD :RT
E0299 >10 FD :FD
E0300 >20 RT :RT
E0302 >30 CIRCLE3 :FD :RT
E0303 >END
E0304 CIRCLE3 DEFINED
E0307 ?CS
E0312 ?CIRCLE3 1 1
E0314 ?CIRCLE3 1 2
E0320 ?CIRCLE3 3 1
```

{Figure 14}

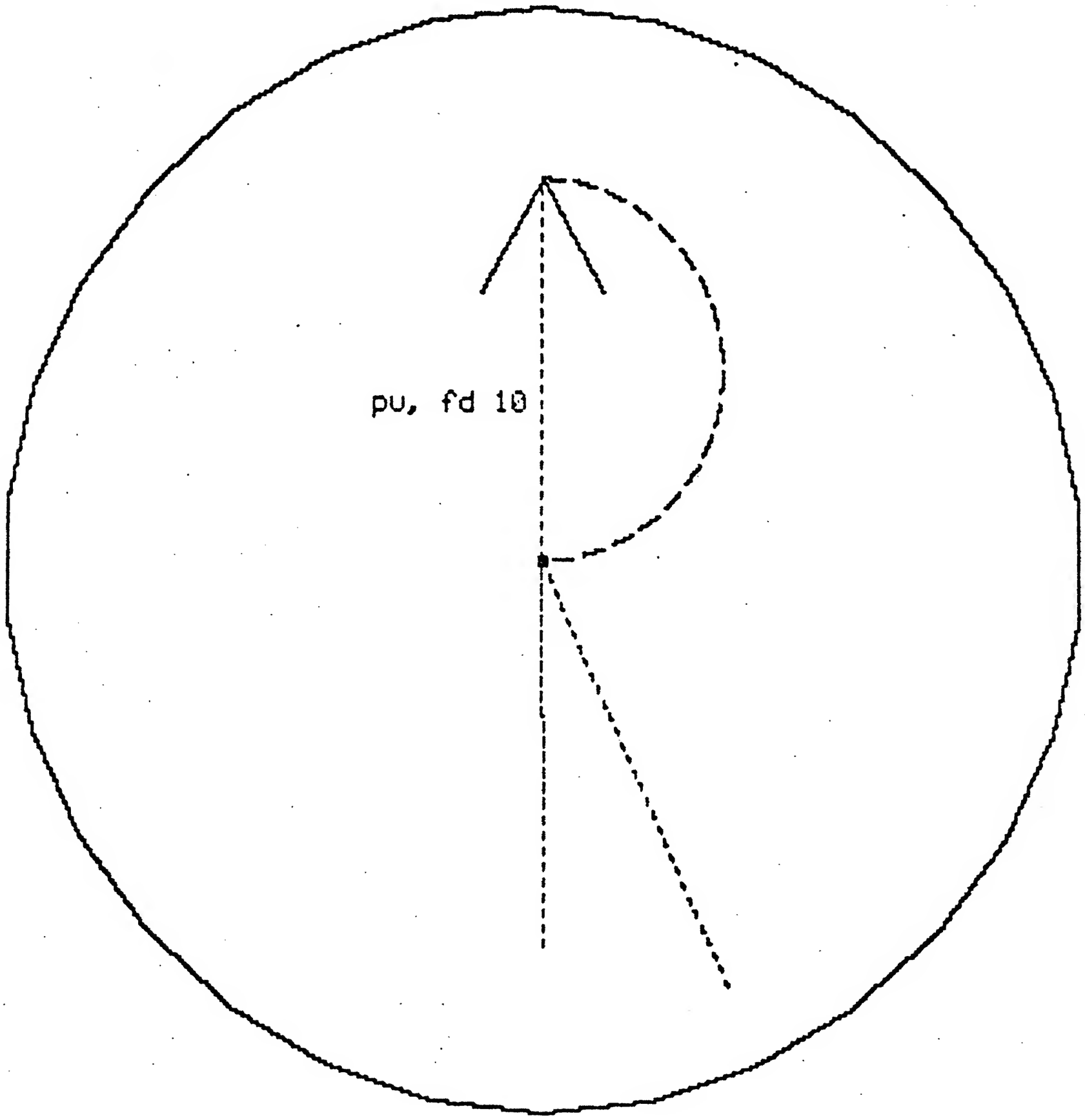
Parsed GREG Protocol, Part Two



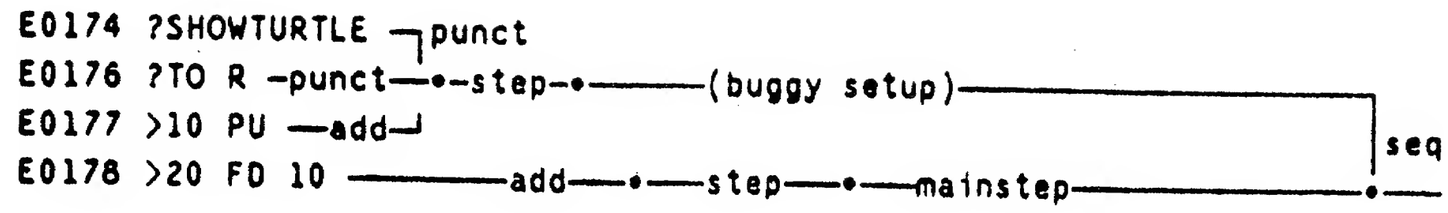
{Figure 15 }

To Draw an R, version 1

(dotted lines show future additions)



{Figure 16}



This is the same bug which was noticed in first defining the G (E0101-E0102). Moreover, it probably had the same origin: the **PENUP** command was a penstate Setup, in preparation for achieving a position Setup which, in fact, was not required. There is no explicitly manifested debugging sequence following the definition of part one of the R; hence it is not surprising that Greg has failed to detect the problem.

The next event is a CLEARSCREEN, which may appear to be an anomaly.

```
E0179 >30 CS
```

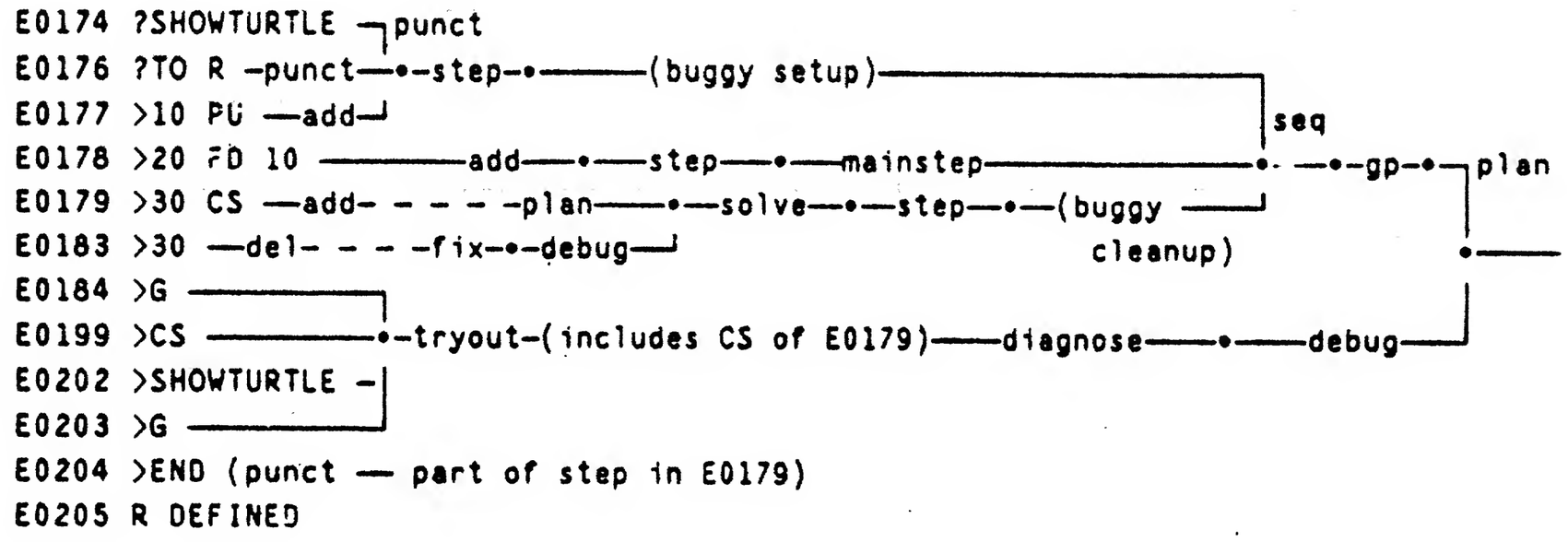
Why include a CLEARSCREEN at this point in the definition, as its effect will be to undo previously accomplished subgoals? What probably happened is that the line number was typed, followed by several minutes of off-line planning by Greg. If so, a breakdown of the elapsed time for the type-in, which was not collected, would have provided an additional clue.

Subsequent events suggest the hypothesis that E0179 was not intended to add a CS to the procedure, but to execute a CS directly (as in fact occurs in Event E0199). Event E0183 undoes the previous type-in.

```
E0183 >30
E0184 >G
E0199 >CS
E0202 >SHOWTURTLE
E0203 >G
E0204 >END
E0205 R DEFINED
```

Events E0184 through E0203[14] are interpreted as part of a Tryout for R (R-MSI), in that they serve to check the initial state for R by observing where G leaves the turtle. According to this view, the CS part of E0179 should be included as part of the Tryout episode, but the line number (30) part should be included as part of a (buggy) Cleanup episode. Since the debugging event for the

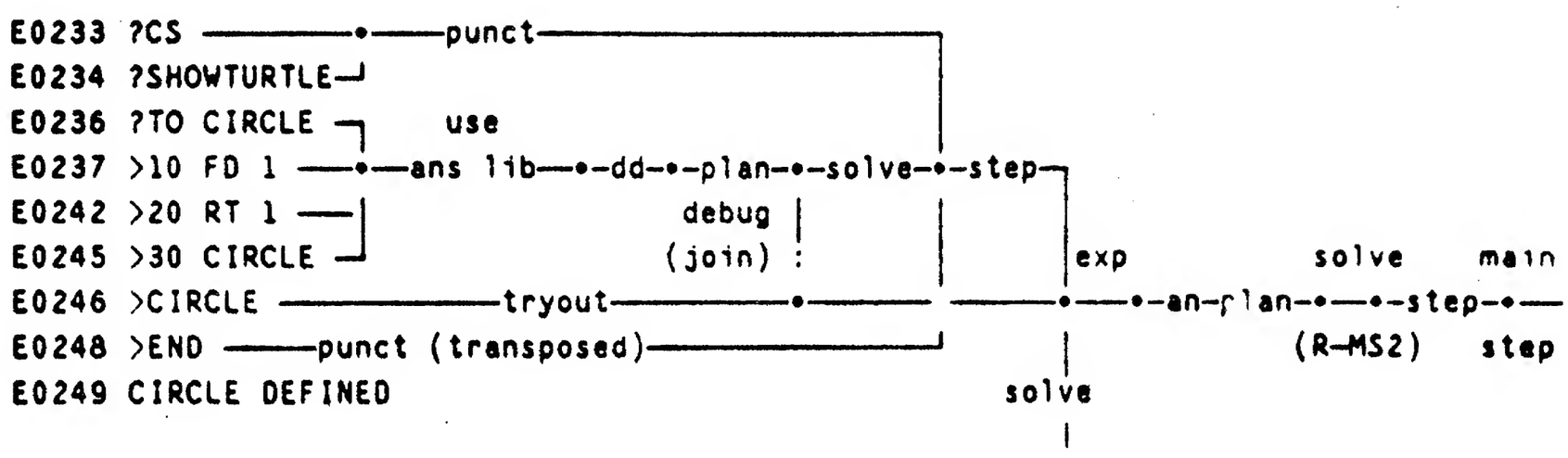
Cleanup episode (E0183) intervenes, this analysis requires a transposition.



At this point, Greg should be about to define the curved portion of the R (labelled R-MS2). If he were pursuing his usual depth first Sequential strategy, he should SOLVE for this Main Step now rather than simply leaving it as an undefined subprocedure and completing the super-procedure for the R. But instead of defining R-MS2 or the rest of R, Greg constructs a CIRCLE program! What is he doing?

```
E0233 ?CS
E0234 ?SHOWTURTLE
E0236 ?TO CIRCLE
E0237 >10 FD 1
E0242 >20 RT 1
E0245 >30 CIRCLE
E0248 >END
E0249 CIRCLE DEFINED
```

We would maintain that Greg is applying an Anticipatory Plan at this point. An Experiment is devised, which involves constructing a *related* procedure -- to draw circles -- and then trying it out. He utilizes a Round Plan to accomplish his goal. This commences an evolutionary sequence in which the knowledge acquired from successive auxiliary problems is later employed in solving the original problem.



What is ultimately taken from the procedure is its Plan to draw arcs by means of piecewise linear approximations. This adaptation of its Domain Dependent Plan is accomplished by experimenting with successively modified versions of the CIRCLE procedure. As first typed in, CIRCLE had spelling bugs, which are not presented here. After debugging, CIRCLE draws {Figure 17}.

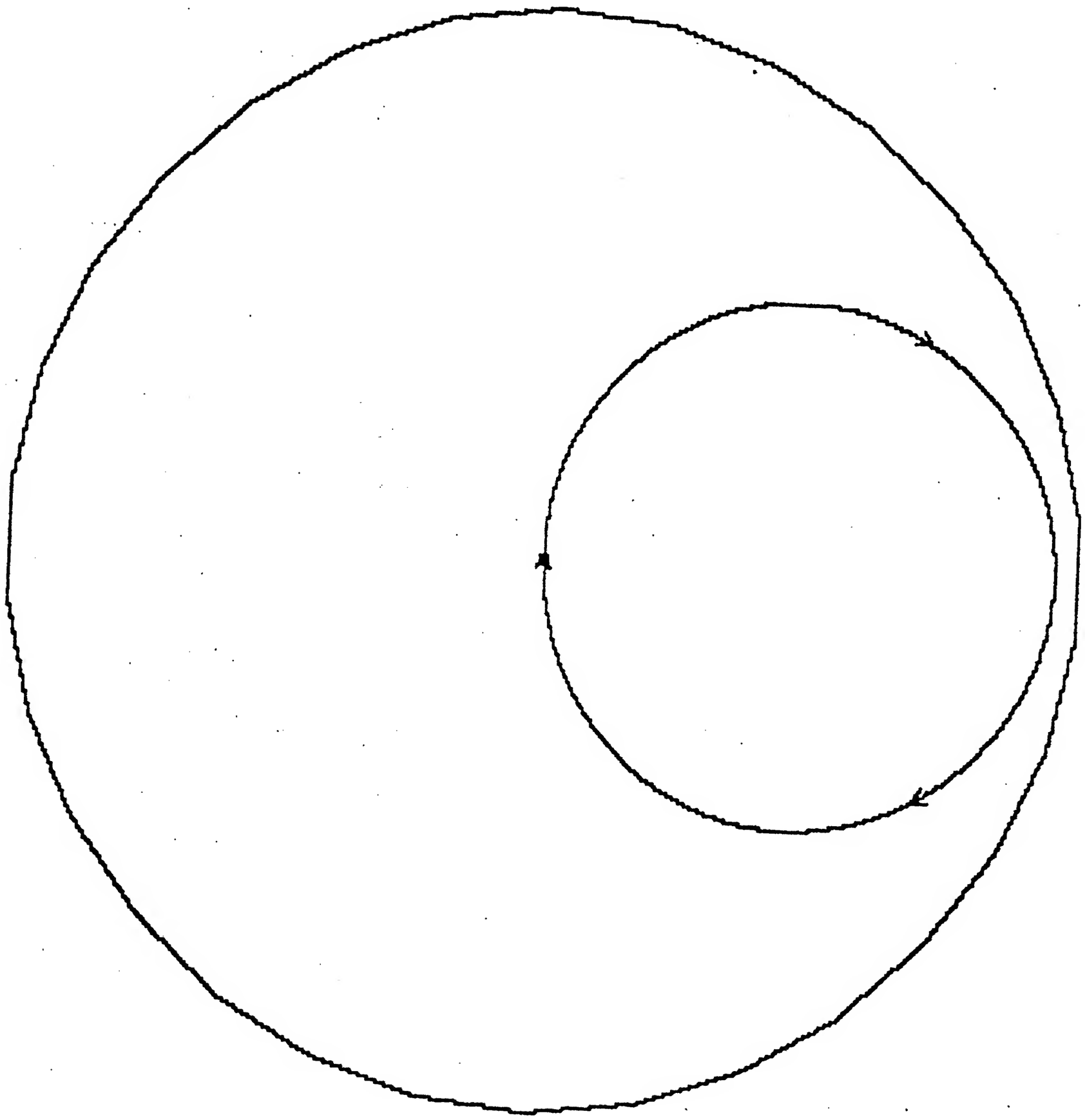
The circle drawn is too large for the corresponding part of the R. The size of the circle is adapted in a succession of new versions, a strategy which ensures that the existing circle will not be harmed by the changes.

```
E0283 ?TO CIRCLE2
E0284 >10 FD .5
E0290 >20 RT .5
E0291 >30 CIRCLE2
E0292 >CS
E0293 >END
E0294 CIRCLE2 DEFINED
E0295 ?CIRCLE2
```

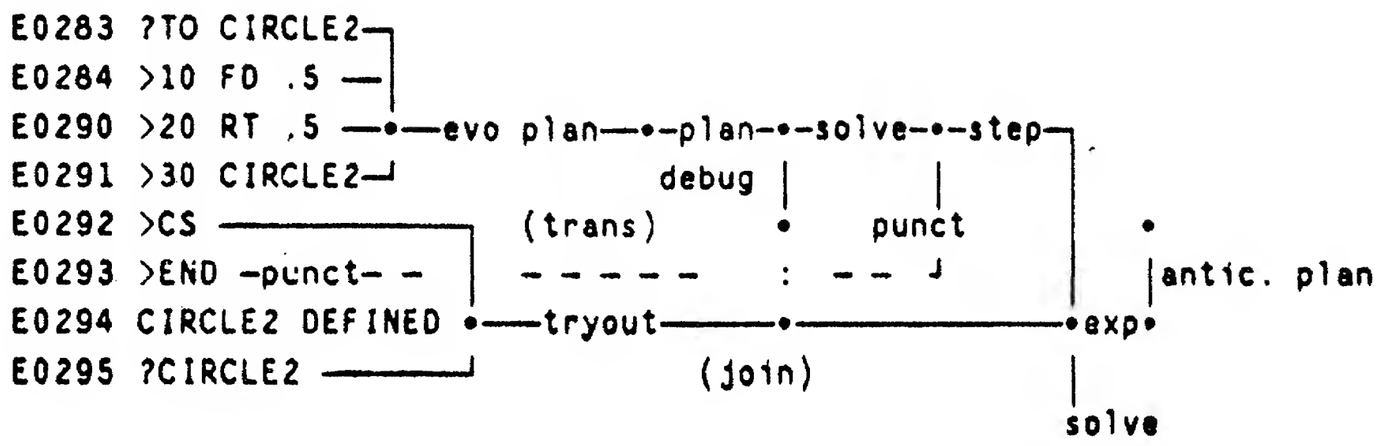
The Plan of CIRCLE evolves into that of CIRCLE2, but Greg's adaptation, based on an analogy with standard uses of scale factors, has a theory bug. Scaling both the rotation and the movement will not change the total size of the circle. Note that the Tryout events (such as E0295) are really serving two purposes: the auxiliary procedure is being debugged to meet its own specifications, and used as part of an Experiment prior to another Solve. Consequently these events appear as "joins" in the figure.

To Circle, first attempt

(too big for an R)



{Figure 17}



An unfortunate fact about current Logo is that the *interesting* bugs are not always the ones which are tackled by the student. Before manifesting the theory bug, Logo first manifested several less enlightening bugs (which are not shown here) related to the use of fixed rather than floating point arithmetic. If the computer learning environment were intelligent enough to recognize such situations, it could increase the frequency with which *pedagogically valuable* bugs were encountered. Here, for example, an intelligent tutoring module could interject queries about the predicted effects of *doubling* the scale factor, perhaps leading to an example which *would* illustrate the basic theory misunderstanding, but avoid the idiosyncrasies of Logo arithmetic conventions. The human tutor, in this case, allowed events to take their course.

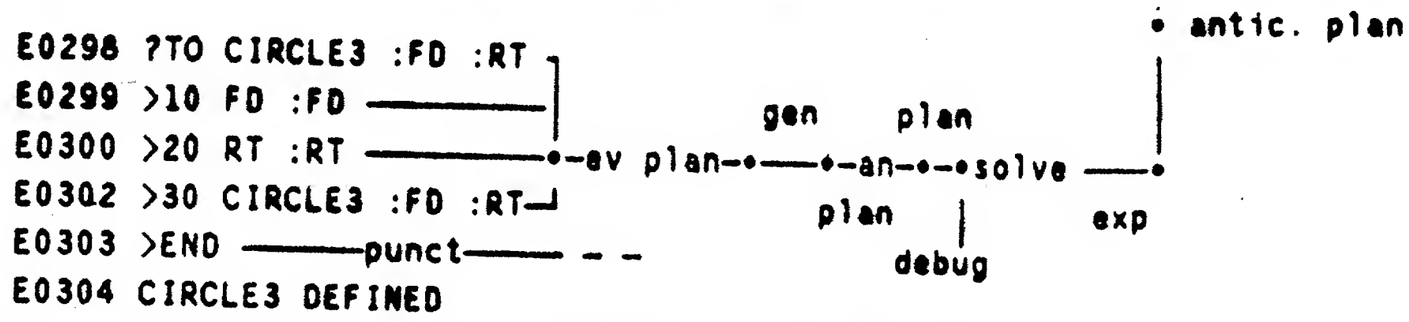
The production of the semi-circle program (R-MS2) by successively modifying the circle (not all of the versions of which are shown here) is prototypical of our notion of an "evolutionary sequence," involving a tight interaction between Anticipatory and Evolutionary Plans. Each version was written, not for its own intrinsic interest, but purely as an auxiliary problem anticipating the further development. The generation of each version was accomplished by a Plan which takes the *previous* version as a pattern for substitution or adaptation.

Insufficient knowledge of the domain brought Greg to an impasse. His expectations about the modified circle were wrong. Before attempting to further debug the *procedure*, Greg needed to debug his theory. In order to induce an alternative theory of circles, Greg wished to experiment with a wider variety of inputs to FORWARD and RIGHT. Events E0298 to E0304 involve the definition of another auxiliary procedure as a subgoal of an Experiment type Anticipatory Plan.

```
E0298 ?TO CIRCLE3 :FD :RT
E0299 >10 FD :FD
E0300 >20 RT :RT
E0302 >30 CIRCLE3 :FD :RT
```

```
E0303 >END
E0304 CIRCLE3 DEFINED
```

The Experiment turns out to yield tangible results. In order to explore a wider range of possible circle programs, the auxiliary procedure was constructed. The auxiliary procedure, which was built as an *experimental tool* for the express purpose of "making it easier to try out different inputs to FD and RT," turned out to *be* the *general case* of the previous problem. Consequently, the generation of this procedure was the appropriate next step in the evolutionary sequence.



This was an exciting discovery for Greg, an "AHA experience." The feeling of elegance apparent in this episode seems to arise from the fact that the reasoning process led to the generation of a single procedure which simultaneously served two distinct purposes. In E0307 through E0322, the Tryout phase of the Experiment proceeds smoothly. Not only is the routine appropriate for testing the limiting cases of the domain primitives, it suggests the patch for Greg's internal theory bug.

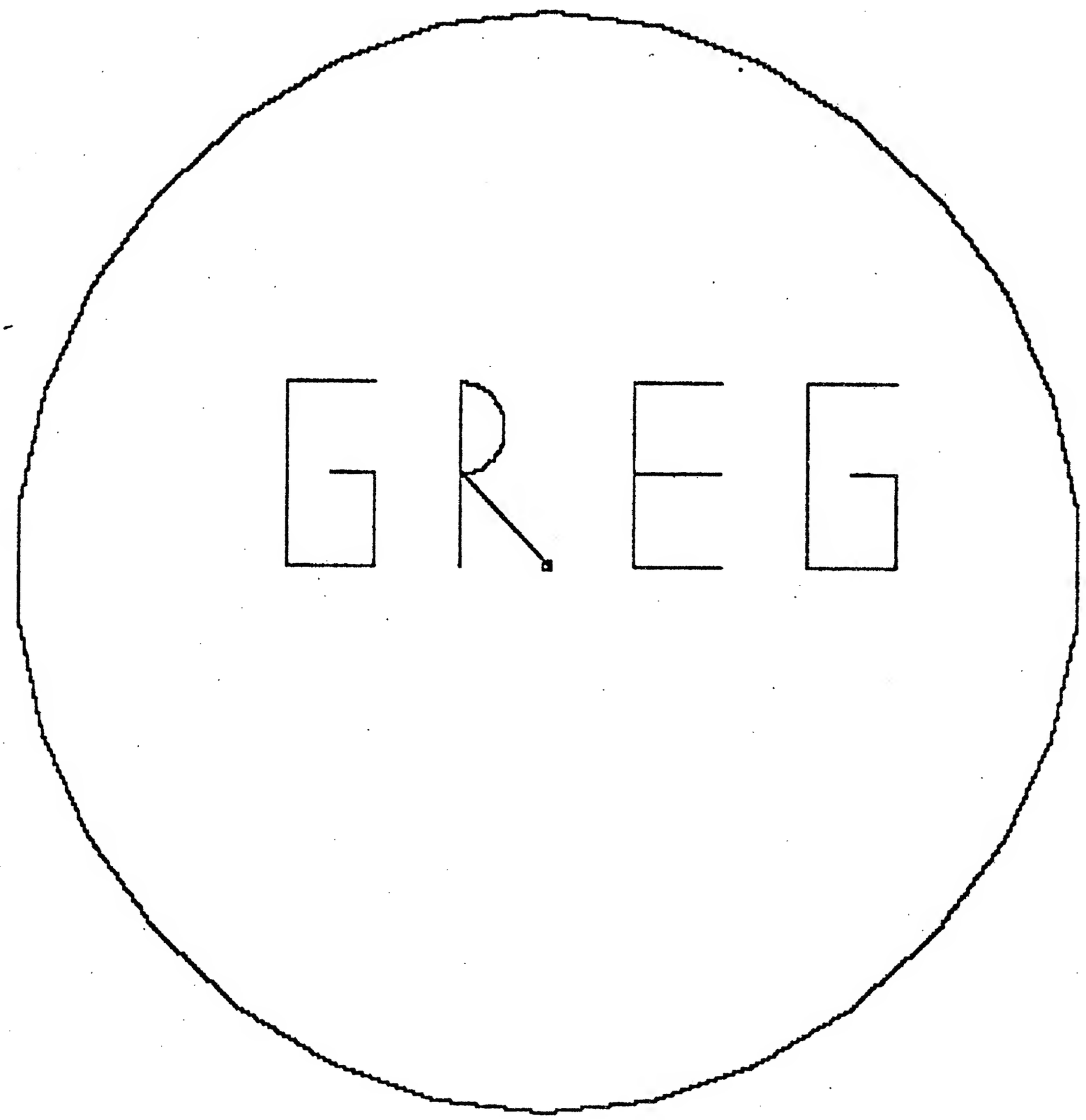
```
E0307 ?CS
E0312 ?CIRCLE3 1 1
E0314 ?CIRCLE3 1 2
E0316 ?CIRCLE3 2 2
E0318 ?CIRCLE3 1 3
```

The CIRCLE3 episode is a convenient place to conclude the analysis of this protocol. In the subsequent episodes, the circle evolves into a general ARC program, which is used to draw the semi-circle, which in turn is used to draw the R. An E procedure is written, and finally GREG itself is written and debugged. {Figure 18} shows the debugged GREG as eventually completed.

Although the notion of protocol parsing developed here is primarily descriptive, some simple testable predictions are provided. For example, one would expect that instances of Plans which are initiated but whose goals have *not* been satisfied should be reactivated when the subplans which they have invoked have terminated. In every case this simple prediction is borne out by the more complete records of this session.

To GREG

(As Eventually Finished)



{Figure 18}

7. Conclusion

> The fundamental hypothesis of genetic epistemology is that there is a parallelism between the progress made in the logical and rational organization of knowledge and the corresponding formative psychological processes. Well, now, if that is our hypothesis, what will be our field of study? Of course the most fruitful, most obvious field of study would be reconstituting human history -- the history of human thinking in prehistoric man. Unfortunately, we are not very well informed about the psychology of Neanderthal man or about the psychology of *Homo siniensis* of Teilhard de Chardin. Since this field of biogenesis is not available to us, we shall do as biologists do and turn to ontogenesis. Nothing could be more accessible to study than the ontogenesis of these notions. There are children all around us. It is with children that we have the best chance of studying the development of logical knowledge, mathematical knowledge, physical knowledge, and so forth.
>
> [Piaget 1971]

Programs and protocols, which by themselves have a very unclear structure, become easier to understand when an hierarchical explanation is imposed. This explanation reveals the goal structure, plans, and bugs of the problem solving process. Such an explanation can be generated by parsing the protocol with a problem solving grammar. We believe that these explanations represent, at least to some extent, decisions being made by the human problem solver. This belief is supported partly by theoretical analyses of planning and debugging, and partly by a detailed analysis of the protocol behavior. Further experimental work is necessary to provide clear evidence for our grammatical approach as a theory of this aspect of human problem solving.

The context-free form which the grammar assumed for parsing the example programs and protocol was a technique for summarizing certain insights into the structure of planning and debugging. Context free grammars, however, are limited and do not capture certain generalities. An example of the formal inadequacy of context free grammar for our purposes is its inability to specify parameters to plans: a rule that introduces a SOLVE recursively should express the fact that its argument is a subproblem. Hence, the sort of *structural* protocol analysis provided by the grammar needs to be supplemented by corresponding *semantic* and *pragmatic* annotation. The semantic annotation would describe the particular problem; the pragmatic commentary would describe why a particular planning strategy was chosen over other alternatives. This formalization

is undertaken in [Goldstein & Miller 1976b].

The analysis procedure should be at least partially automated, to improve the practicability and objectivity of the enterprise. With a formalization of semantic and pragmatic problem solving knowledge to supplement the basic grammar, this becomes possible. [Miller & Goldstein 1976d] describes the *design* of an automated parser. When implemented, such a protocol analysis system should be valuable as an experimental crucible for our theory of problem solving, and as a module for incorporation into intelligent tutors [Goldstein & Miller 1976a]. In the latter case, the parsed protocol would constitute a database from which the tutor could abstract a description of the student's problem solving strategies: those being successfully applied, as well as those being ignored or applied incorrectly.

Finally, there is another way of testing the power of our problem solving grammar. This is to embed it in an editor for defining and modifying programs. In such an environment, there is no need to guess which planning rule is being applied by the programmer: the choice is made explicitly by the user. While this editor will not reveal by itself whether our theory is psychologically valid, it will indicate whether an individual can be comfortable within the problem solving confines dictated by the grammar. We have designed such an editor [Miller & Goldstein 1976c] and plan to experiment with its utility as an assistant to a human problem solver.

The construction of computational models of human cognition is an ambitious undertaking. An approach based on the use of concepts from computational linguistics -- grammmars, semantics, pragmatics, parsing -- appears to be profitable. To justify this statement, the next phase of our research will involve a series of experiments in AI (is the theory sufficient to support competent problem solving); in psychology (do protocols reveal the structure suggested by the theory, can individuals be modelled by perturbed versions of the grammar); and in education (does exposure to the theory improve an individual's problem solving competence).

8. Notes

1. To be of any use, a Plan or problem decomposition strategy must include "recomposition" information (i.e., advice on how the solutions to the subproblems are to be recombined to obtain a complete solution to the original problem). Moreover, it should be stressed that we recognize other aspects to the problem solving process besides planning. For example, certain coherent techniques for gathering information are essential for "exploring the problem space." Similarly, methods for "debugging" an "almost-right Plan" are essential for bringing a planning process to fruition. Without debugging techniques, the resulting requirement that a Plan bring complete success would be far too restrictive. Debugging is analyzed in section 4, [Goldstein 1974], as well as in [Goldstein & Miller 1976b] and [Miller & Goldstein 1976c].

2. We present an early version of the taxonomy and planning grammar, reflecting the state of our understanding as of the original writing of this essay. In later papers we revise many of the details. Nevertheless, we adhere both to the appropriateness of a formalism based upon a grammatical analogy, and to the approximate content of the rules. In short, we have not revised our primary hypothesis, that a theory of this sort can account for a wide range of problem solving phenomena. The evolution of our theory is further explained in [Miller & Goldstein 1976a].

3. Projects are the normal state of affairs in real-life problem solving, rather than the alternative paradigm of isolated problems suggested by the typical textbook.

4. This view of planning is a simplification. It asserts that the problem is analyzed in a top down fashion. Of course, the problem solver can engage in exploration and experimentation; or can identify a subgoal without having a clear understanding of the overall plan. The dynamics of exploration are not formalized by this grammar.

5. Our use of a context free grammar for problem solving closely resembles D. Rumelhart's [1975] work on story grammars. It should be interesting to see to what extent our respective theories, designed to account for superficially very different phenomena, continue to develop in parallel.

6. The rules of the grammar are written using the following syntax:

disjunction: "a | b" is read as, "a or b".
conjunction: "a + b" is read as, "a and b".
optionality: "(a)" is read as, "a is optional"
iteration: "⟨a⟩*" is read as, "a repeated one or more times".

7. Some events which are treated as "punctuation" by these rules might be further analyzed as part of a "secretarial process" which itself was described by a slight extension of the grammar. For example, the TO and END punctuation might be classified as Setup and Cleanup steps in a ***Plan to define*** the procedure. Additionally, the structure of the problem solver *per se*, and perhaps even the ***development*** of the problem solver might be describable using such a grammar. Such generalizations would be encouraging, but are merely speculation at this point.

8. Note that TRYOUT now occurs on the right hand side of two rules. It occurred in the grammar rule P17 describing an experiment as well as in this grammar rule describing diagnosis. Hence, the problem arises in analyzing a protocol in deciding which purpose a given tryout is intended to accomplish -- an anticipatory experiment or a debugging test. The grammar is non-deterministic and does not decide this issue. Additional semantic and pragmatic constraints are necessary. For example, has a program just been completed and is the tryout an execution of this program. On the other hand, is the tryout an execution of a previously debugged program in some new context. The former suggests diagnosis: the latter experimentation. In generating the parse of sections 5 & 6, we apply this kind of criteria to choose between alternative interpretations allowed by the grammar. Whenever this is done, we cite the basis of our decision. In [Goldstein & Miller 1976b] some of these additional constraints are formalized.

9. Our hypothesis is that a full accounting for the problem solving process will involve at least three distinct aspects: structural, semantic, and pragmatic. We find it fruitful to think of the structural component as a specification of the ***potential control paths*** in a procedural problem solving system. The framework provided here addresses itself only to this structural component. The data flow and branching conditions are ***not*** specified. (The extent to which it is possible to study a given component separately from the others, as has been attempted here, is an empirical question.) The partial theory presented in this essay -- one describing only the structural component -- can provide, at best, only a partial account: one which is ***inherently nondeterministic.*** That is, while the theory ***imposes constraints*** on the allowable behaviors, it is ***unable in principle*** to predict precisely what the student will do next. This is entirely analogous to the constraints on natural language utterances imposed by the syntactic component of a generative grammar in modern linguistic theory.

10. A more detailed comparison of production systems to our own approach, as applied to automatic protocol analysis, is undertaken in [Miller & Goldstein 1976d].

11. The protocol is data from an actual session with a single student, abridged only by deleting lines which were not essential for analyzing the underlying problem solving behavior. Typically these deletions involved the use of the BREAK character to delete a mistyped line, lengthy PRINTOUTS of procedures, system errors, and less interesting episodes involving spelling mistakes and the like. The complete, unmodified version of this file, as well as several similar files, are available from the authors in machine readable form. Omissions are indicated by gaps in the Event numbering (left most, beginning with "E").

12. We have tolerated a few instances of two minor deviations from a strict context free derivation. When a single event seemed to serve two distinct purposes (such as both deleting a previous line of code, and adding a new one), we have allowed "joins," which are analogous to copying an event so that it appears twice in the parse tree. When an event seemed to be out of order with respect to episode grouping (such as an END which comes after the defined procedure has been tried out), we have allowed "transpositions," which are shown as crossed lines in the diagram. Both of these violations could be avoided by slightly altering the grammar. However, it would be more parsimonious to assume a simple transformational component. In order to improve the page layout, we have suppressed some low level details. In all other respects, the figures represent a correct derivation in our grammar.

13. One way to experiment with the nature of the analyses generated by the grammar is to define a set of summarization rules. These rules take a parsed protocol and generate a summary. This validation technique (e.g., does the summary seem reasonable to an unbiased human teacher?) is modelled upon Rumelhart's use of the technique for testing his grammatical theory of stories. We have already seen one example of summarization, namely, generating a collapsed plan of the final program. Other kinds of summarization rules might accept a parse as input, and return as output a list of the plan-types or diagnostic strategies which were employed.

14. The deleted lines here were due a minor hardware failure: intermittent printing errors local to the computer terminal.

9. References

[Chomsky 1957]

Chomsky, Noam, *Syntactic Structures*, The Netherlands, Mouton (Eighth printing, 1969), 1957.

[Goldstein 1974]

Goldstein, Ira P., *Understanding Simple Picture Programs*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 294, September 1974.

[Goldstein & Miller 1976a]

Goldstein, Ira P., and Mark L. Miller, *AI Based Personal Learning Environments: Directions for Long Term Research*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 384 (Logo Memo 31), December 1976a.

[Goldstein & Miller 1976b]

Goldstein, Ira P., and Mark L. Miller, *Structured Planning and Debugging: A Linguistic Theory of Design*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 387 (Logo Memo 34), December 1976b.

[Hewitt 1972]

Hewitt, Carl, *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 258, April 1972.

[Miller et al. 1960]

Miller, George A., Eugene Galanter and Karl H. Pribram, *Plans and the Structure of Behavior*, New York, Holt, Rinehart, and Winston, 1960.

[Miller 1976]

Miller, Mark L., *Cognitive and Pedagogical Considerations for a Tutorial LOGO Monitor: An Investigation Into the Evolution of Procedural Knowledge* (Master's Thesis), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February, 1976.

[Miller & Goldstein 1976a]

Miller, Mark L., and Ira P. Goldstein, *Overview of a Linguistic Theory of Design*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 383 (Logo Memo 30), December 1976a.

[Miller & Goldstein 1976c]

Miller, Mark L., and Ira P. Goldstein, *SPADE: A Grammar Based Editor for Planning and Debugging Programs*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 386 (Logo Memo 33), December 1976c.

[Miller & Goldstein 1976d]

Miller, Mark L., and Ira P. Goldstein, *PAZATN: A Linguistic Approach to Automatic Analysis of Elementary Programming Protocols*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 388 (Logo Memo 35), December 1976d.

[Newell 1966]

Newell, Allen, *On the Analysis of Human Problem Solving Protocols*, Carnegie Institute of Technology, paper for International Symposium on Mathematical and Computational Methods in the Social Sciences, Rome, July 1966.

[Newell & Simon 1972]

Newell, Allen, and Herbert Simon, *Human Problem Solving*, Englewood Cliffs, New Jersey, Prentice-Hall, 1972.

[Piaget 1971]

Piaget, Jean, *Genetic Epistemology* (trans. Eleanor Duckworth), New York, W.W. Norton, 1971.

[Polya 1962]

Polya, George, *Mathematical Discovery* (Volume 1), New York, John Wiley and Sons, 1962.

[Rumelhart 1975]

Rumelhart, David E., "Notes on a Schema for Stories," in D. Bobrow & A. Collins, *Representation and Understanding: Studies in Cognitive Science*, New York, Academic Press, 1975, pp. 211-236.

[Solomon 1976]

Solomon, Cynthia J. , *A Case Study of a Young Child Doing Turtle Graphics in Logo*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 375 (Logo Memo 28), July 1976.

[Sussman 1973]

Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 297, 1973.

Figures

MILLER & GOLDSTEIN

December 27, 1976

TJ6-able Source File Was: DSK:MILLER;GREC 349 — 350 has improved title page

This Document Was Produced Using: TJ6 23

Do NOT Duplicate this page